NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DISTRIBUTED ARCHITECTURE FOR THE OBJECT-ORIENTED METHOD FOR INTEROPERABILITY

by

George M. Lawler

March 2003

Thesis Advisor: Valdis Berzins
Second Reader: Paul Young

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Distributed Architecture for the Object-Oriented Method for Interoperability			**5. FUNDING NUMBERS**
6. AUTHOR(S) George Michael Lawler			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Space and Naval Warfare Systems Center San Diego, CA 92152-5031			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The Department of Defense (DoD) is both challenged by the quest for interoperability and capable of the bottom-up development of a solution. The predominant method for achieving interoperability is the development of an intermediate representation that provides a common integration language or data model. An example is Young's Object-Oriented Method for Interoperability (OOMI), which produces a Federation Interoperability Object Model (FIOM) for the resolution of heterogeneities in representation and view of a real-world entity. An FIOM generates a standard for interoperability by associating the non-standard, component system data models into an extensible lattice, which captures translations that resolve data modeling differences. To support the bottom-up creation of an FIOM we; (1) describe a self-similar approach to data storage that allows generic data structures to be manageable, extensible and asynchronously populated, and (2) introduce a lattice concept for facilitating efficient and scalable object inheritance relationships. We assert that DoD's acquisition environment necessitates a distributed approach to solving the interoperability challenge. We present the description of a distributed software system to facilitate the collaborative construction of an FIOM within the existing DoD structure and provide an architecture to guide the development of such a distributed collaborative environment.

14. SUBJECT TERMS Interoperability, Object-Oriented Method for Interoperability, Distributed Systems, Collaboration, Self-Similar Data Structures, Lattice Concept, FIOM Lattice, Peer Networking, XML, Data Binding, Layered Architecture, Distributed OOMI			**15. NUMBER OF PAGES** 157
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DISTRIBUTED ARCHITECTURE FOR THE OBJECT-ORIENTED METHOD FOR INTEROPERABILITY

George M. Lawler
Lieutenant, United States Navy
B.S., United States Naval Academy, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2003

Author: George M. Lawler

Approved by: Valdis Berzins
Thesis Advisor

CAPT Paul Young, USN
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Department of Defense (DoD) is both challenged by the quest for interoperability and capable of the bottom-up development of a solution. The predominant method for achieving interoperability is the development of an intermediate representation that provides a common integration language or data model. An example is Young's Object-Oriented Method for Interoperability (OOMI), which produces a Federation Interoperability Object Model (FIOM) for the resolution of heterogeneities in representation and view of a real-world entity. An FIOM generates a standard for interoperability by associating the non-standard, component system data models into an extensible lattice, which captures translations that resolve data modeling differences. To support the bottom-up creation of an FIOM we; (1) describe a self-similar approach to data storage that allows generic data structures to be manageable, extensible and asynchronously populated, and (2) introduce a lattice concept for facilitating efficient and scalable object inheritance relationships. We assert that DoD's acquisition environment necessitates a distributed approach to solving the interoperability challenge. We present the description of a distributed software system to facilitate the collaborative construction of an FIOM within the existing DoD structure and provide an architecture to guide the development of such a distributed collaborative environment.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

CCR	Component Class Representation
DOM	Document Object Model
FCR	Federation Class Representation
FE	Federation Entity
FEV	Federation Entity View
FIOM	Federation Interoperability Object Model
GUI	Graphical User Interface
IE	Interoperability Engineer
IDE	Integrated Development Environment
OOMI	Object-Oriented Method for Interoperability
SAX	Simple API for XML
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XML	Extensible Mark-up Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Thank you to CAPT Paul Young for guidance, patients and support during this process, but mostly for the Object-Oriented Method for Interoperability, a solution that I believe will solve interoperability for the DoD.

Thank you to Steve Shedd and ShongCheng Lee for your thoughtful additions to the OOMI and the OOMI IDE prototype. Your efforts paved the way for this thesis.

Thank you to Bret Michael for the incalculable contribution your example and tutelage have made to my graduate education and this thesis.

Thank you to my wife Angela, who agreed on many occasions to have fun later so that this might be done now and for doing a great job at faking an interest in computer science. Thank you to Paige and Brooke for so many smiles, the early wake-ups that increased my daily requirement for caffeine and our trips to the aquarium.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THE PROBLEM

An on going debate concerning the future of the United States Department of Defense (DoD) contains as an element of its core the transformation of the current military establishment into the form that shall be required for the near and distant future. The most recent technological advances have sparked the imaginations of many, but the latest round of changes and reforms are not likely to resolve the quandary concerning the multitude of aging *legacy* combat, communications and information systems that persist in the inventory. Completely replacing every existing system with new systems is risky, financially prohibitive and unnecessary to achieve improved capability. The challenge then is to re-combine the existing systems to produce the desired transformational capabilities, by creating a scalable, flexible and long-term interoperability solution that fits within the current organizational and cultural constraints.

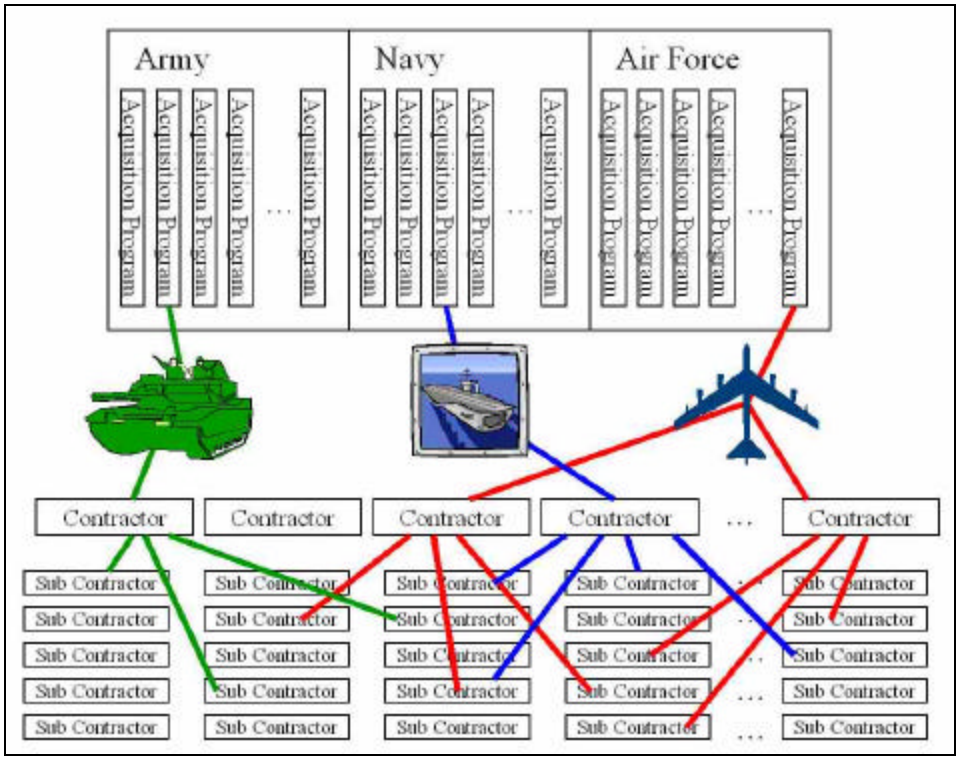

Figure I-1. The Complexity Of Department Of Defense Acquisition.

The acquisition programs and contract organizations that procure and produce the DoD systems are the most likely sources of knowledge concerning the legacy systems and therefore are a great asset in the quest for interoperability. The breadth of DoD responsibilities and functions requires the use of a distributed acquisition structure (notionally presented in Figure I-1), but this hierarchical structure, which continues to be a great strength, is challenged by the urgent requirement to procure interoperable technology systems and produce interoperable combinations of legacy systems. The sheer number of systems in use and under development suggests the conclusion that a coordinated engineering effort to achieve seamless runtime interoperability is an intractable goal. Though a complex and difficult challenge, to be sure, the interoperability solution sought by the DoD is not impossible to achieve. The reason being that despite the distributed environment in which complex systems are developed, at some level each complex system contains an information system that can be made to interoperate. Figure I-2 presents a more abstract view of the interoperability challenge.

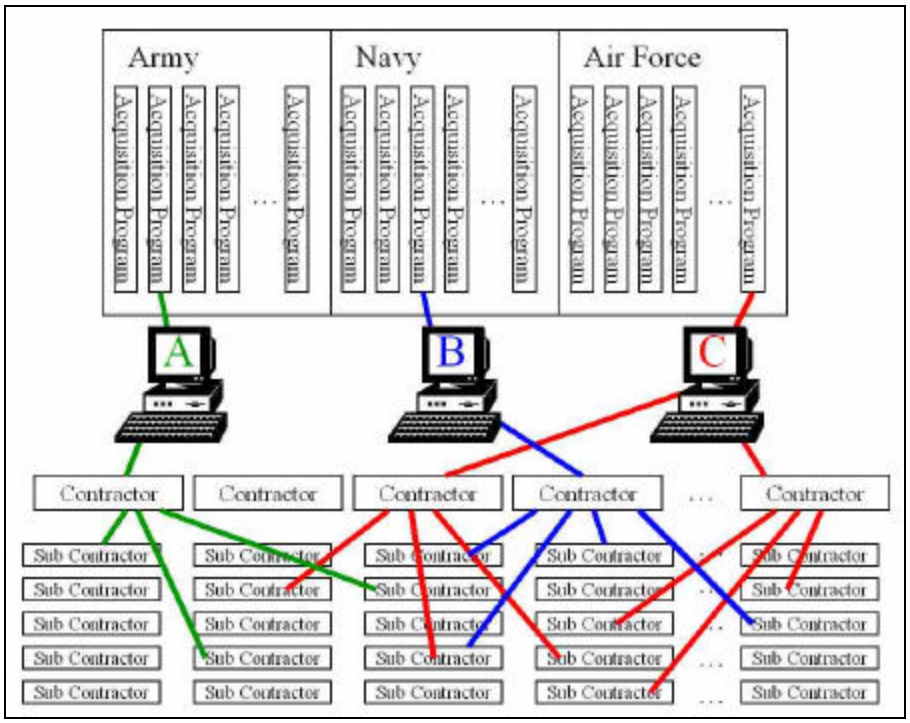

Figure I-2. Simplified View Of DoD Systems Acquisition.

The key to success in sharing information between independently developed systems, as demonstrated by the Internet, involves implementation of a layered architecture of common intermediate languages, which develop a common framework under which all entities can resolve their differences. The layered networking approach, in which each layer defines a standard interface, turns out to be a scalable, dynamic, and extremely cost-effective way to develop and deploy independently developed systems, while maintaining a common framework that promotes maximal participation in information sharing. The one catch is that at some level of the layered abstraction, a single *standard* must be available, by way of which all the systems are ultimately connected. The de facto standard for the Internet is the Internet Protocol (IP) and it became the standard quite by accident based mostly on the popularity and availability of the protocol. Adopting a fixed standard developed in a top down fashion would either restrict the future acquisition of new technologies to maintain interoperability with older systems or require a redefinition of the standard. At the same time, hoping for an accident to discover a de facto standard intermediate representation that facilitates interoperability for DoD is a less than desirable strategy.

One cost effective interoperability solution that can meet DoD's requirements rapidly and for the majority of systems is described by the Object-Oriented Method for Interoperability (OOMI) [You02]. Young's OOMI presents a methodology by which a heterogeneous group of systems are formed into a system federation to resolve differences in modeling. An OOMI Integrated Development Environment (OOMI IDE) lends automation assistance to the pre-runtime construction of a Federation Interoperability Object Model (FIOM). Runtime interoperability is achieved with the use of an OOMI Translator, which uses the FIOM to resolve modeling differences of representation and view between pairs of registered component systems. Young's FIOM is a common intermediate representation for the federation of systems, in effect a model that provides a standard for interoperability, constructed in a bottom-up fashion to provide an exact, yet extensible solution in the terms of systems in use today. Future systems will continue to evolve and current systems will persist for as long as several decades, but the OOMI is extensible to provide resolution for these un-knowable future

differences by allowing a new representation to be related to an old representation and by providing automated assistance in defining the relation. The flexibility of the OOMI solution is preferable in the DoD interoperability domain, but the OOMI is not available for use in the DoD environment, at present. To use the OOMI, a DoD-wide collaboration must first be enabled. This collaboration mechanism is described by this research and its proposed architecture, titled **Distributed OOMI**. The step from where we are today to the deployment of a collaborative software toolset based on Distributed OOMI is feasible. The enabler that allows this first step to interoperability is the proven effectiveness of Internet technologies and the availability of Internet infrastructure, more or less pervasively, throughout the DoD.

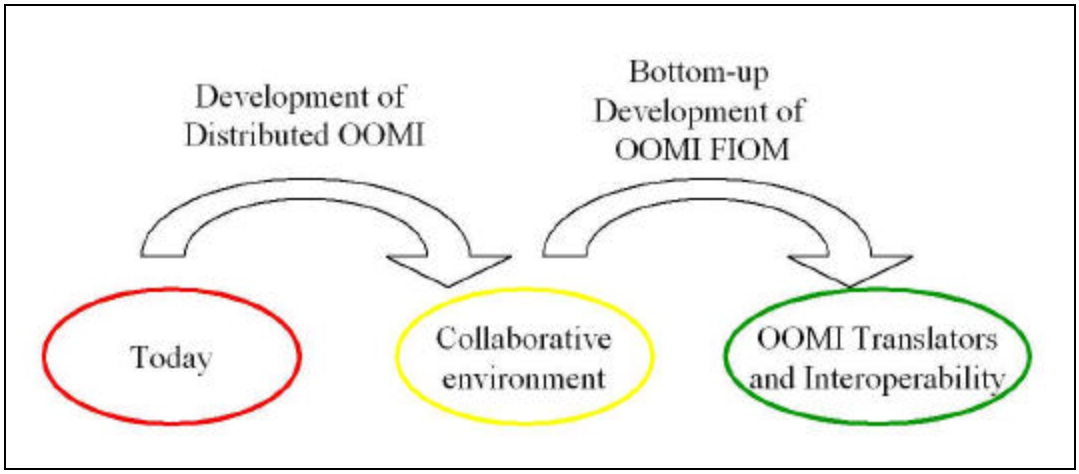

Figure I-3. The Path To Interoperability.

The problem reduces to the need to define a distributed system architecture to support collaborative creation of an FIOM according to the processes defined by the OOMI. The resulting architecture will guide the design effort that will lead to the implementation of collaborative software tools that reduce the complexity of the DoD Acquisition environment to facilitate the development of the required interoperability solution, under the OOMI. The large number of DoD systems in question will require an incremental creation of the solution, and persistence of the overall model for several decades, indeed perhaps indefinitely.

B. THE ARCHITECTURE

The desired distributed system architecture is intended to guide the software development efforts that produce the distributed system that will enable collaborative construction of the FIOM. As a guide, the Distributed OOMI architecture out of necessity must represent complex implementations in abstract and general terms. Choosing good abstractions is the key to developing good models, and developing a model for a distributed architecture that persistently stores an FIOM is not an exception. Because an FIOM is itself a complicated model, we must decide on a simplification of the FIOM for use in describing the architecture in general terms.

The physical and logical location and relation between the distributed application and the storage solution are concepts that support the collaboration, are not trivial and are discussed in terms of network infrastructure currently available. The goal of this thesis is to provide an architecture for use in a peer-to-peer collaboration via a distributed data storage solution, yet we assume that this solution is to be implemented on top of a network infrastructure not necessarily well suited to support distributed storage. While the details of the location of distributed components are meant to be hidden from the application user, a well-structured discussion of these details is imperative to the design of the solution. A layered architecture provides the most flexibility for implementation and future extension, and so we sub-divide the overall abstraction of the solution into three layers, shown in Figure I-4 as the Application, Storage and Peer Networking sections of the architecture.

The Application Layer represents an instance of an OOMI IDE (Integrated Development Environment), the software tool described by the OOMI that facilitates construction of the FIOM [You02]. Prior to Distributed OOMI, the only instance of an OOMI IDE provided enough functional capability to build a rudimentary FIOM but did not provide persistent distributed storage or collaboration. Developing a distributed storage solution is paramount to creating an instance of an OOMI IDE that will support distributed collaboration.

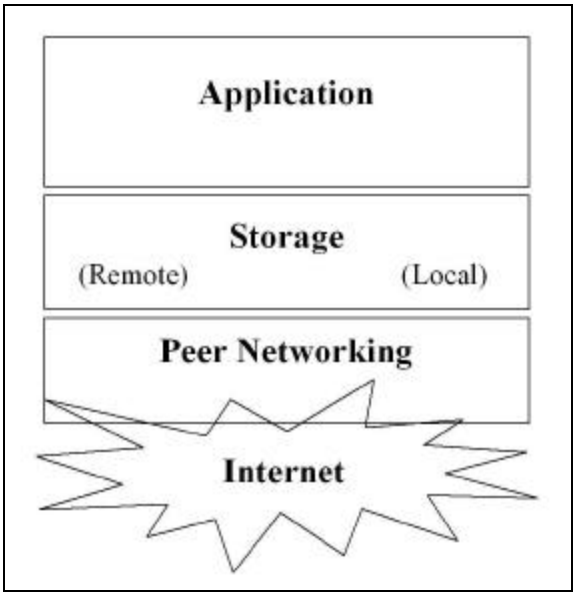

Figure I-4. Distributed OOMI Layered Architecture.

The Storage Layer encompasses the data structures and functional requirements of distributing the FIOM in persistent storage in order to maintain scalability, openness and extensibility. Two types of storage, Remote and Local, partition the storage domain and provide an asynchronous method of sharing the large amount of information required to represent the FIOM, while minimizing the requirements on the lower levels.

The peer-to-peer collaboration is an important and developing concept that extends the existing Internet by allowing conceptual context to be applied to sets of network activity that were previously independent. Context in network computing enables a combination of complex communications to be organized into a single collaborative environment. This functional capability is not available in the current Internet, and so the architecture provides its own networking layer on top of the Internet layer. The Internet then is used for communications between Distributed OOMI participants, with no requirement except to perform its primary function of routing data packets from a known source to a known destination.

C. THE APPROACH

By pushing the processing requirements to the network and up the networking stack to a position above IP, we leverage the entire capability of the current Internet

without requiring additional new infrastructure and without making unrealistic demands of the existing infrastructure. The resulting description of a distributed system is useful in creating a collaborative software environment that will serve as a common ground across the DoD Acquisition environment to lesson the complexity of the required cooperative interoperability effort. As depicted in Figure I-5, the Distributed OOMI architecture describes a solution that provides for the target environment a means to collaborate horizontally within the established hierarchical structure and without reorganization. The collaboration must occur in the middle ground of Defense Acquisition, between the programs and the contractors, because this is the level where the knowledge of the systems exists, and hence conceptually, where the systems themselves exist.

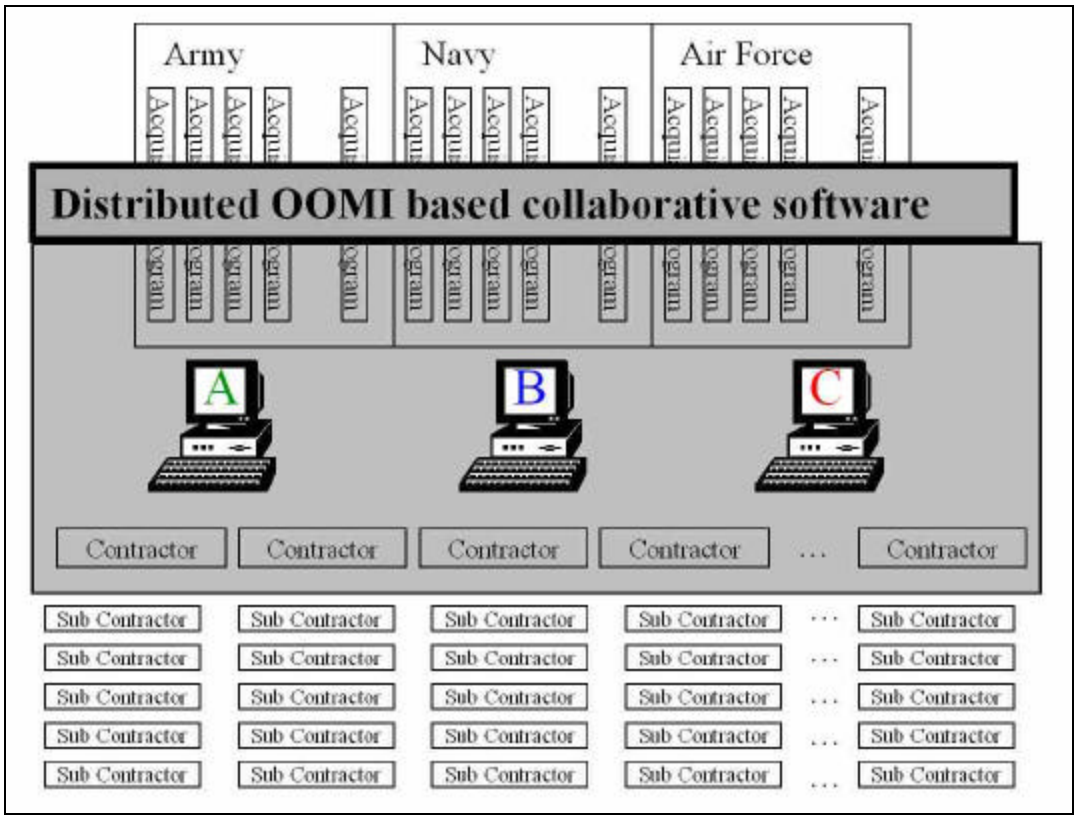

Figure I-5. Enabling Collaboration In A Distributed Environment.

The DoD programs and contractors continue to operate independently in the development of systems according to the procedures and practices most appropriate to

their respective problem domains. At the appropriate time in the acquisition process, each system is independently registered in the FIOM, under the OOMI. Over time, the Distributed OOMI enabled collaboration results in the bottom-up creation of a standard intermediate model to be used for the realization of interoperability between all registered systems. The FIOM thereby exists coincident with the systems and is extensible in the future through the continued registration of new systems.

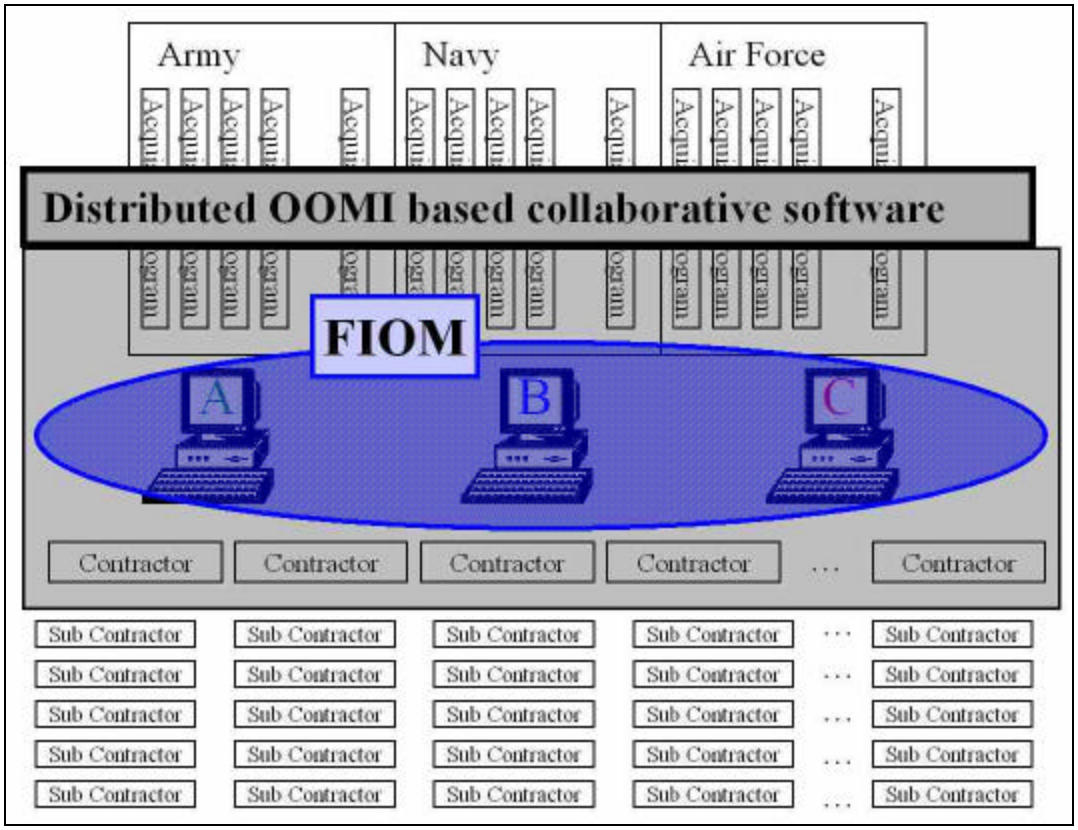

Figure I-6. Distributed OOMI Facilitates Federation Interoperability Object Model (FIOM) Creation, Maintenance And Use.

We present, therefore, a solution to the first step of the challenge (Figure I-3) of building a distributed interoperability model. The Object-Oriented Method for Interoperability (OOMI) provides the common intermediate language required to facilitate communication between dissimilar DoD systems. The Federation Interoperability Object Model (FIOM) is a *standard* developed from the bottom-up, and is completely extensible to meet future requirements without redefinition. The wrapper-based OOMI Translator described under the OOMI uses the FIOM to resolve differences

between incompatible systems to enable information exchange and joint task execution. The Internet already provides physical connectivity between DoD systems in many cases, and therefore is the necessary conduit for collaborating in the construction of the FIOM, as shown in Figure I-6. When the OOMI Translator is made available to each DoD system registered in the Federation, the systems will have the common intermediate representation required to interoperate via the existing network infrastructure technology used by the Internet. All of this is made possible by the Distributed OOMI architecture, which describes a distributed system approach that facilitates the use of the OOMI within the DoD acquisition environment.

We derive our solution from an aggregation of existing approaches to both the OOMI FIOM and distributed systems in general. Chapter II begins with a review of the basics of the OOMI and FIOM construction to familiarize the reader with the first half of the problem space. This chapter also introduces the basics of distributed systems, reviews popular architectural styles for software and distribution and finally describes a method by which distributed systems may be compared and contrasted.

Chapter III describes the Distributed OOMI Architecture in detail and assigns a classification to our approach, so that comparison may be made to other distributed systems. Chapter IV presents our thoughts on implementation of an OOMI IDE based on Distributed OOMI.

Chapter V provides a view of the future that makes use of Distributed OOMI to solve DoD interoperability. Open issues for future research are enumerated and a brief summary and discussion of conclusions learned during the development effort are discussed as well.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

The scope of the problem addressed by this research includes two complex areas of computer science, the first being interoperability and the second being distributed computing. Out of necessity, the first half of the problem domain, the interoperability portion, will review only one method. The case for and against other methods of interoperability is made by Young in [You02] and the reader is directed to his work for the discussion of interoperability in general. To familiarize the reader with Young's proposed method, Section II.A reviews his Object-Oriented Method for Interoperability (OOMI), specifically focusing on the construction of a Federation Interoperability Object Model (FIOM). The second half of the problem space is introduced with a review of the concepts common to all distributed systems and the classic architectures for distribution in Sections II.B and II.C. Lastly, Section II.D introduces a model that provides the means for comparing and contrasting specific distributed implementations and includes the classification of several example systems.

A. OBJECT ORIENTED METHOD FOR INTEROPERABILITY

When considering the complex subject of systems interoperability it is best to establish a clear frame of reference. Of primary concern is the identification of the set of systems considered for participation in an interoperability effort. One possibility involves the integration of multiple systems that pursue a common objective and share the same view of the problem domain. For these *homogeneous* integration efforts the result is an *integrated system.* Failure occurs for this type of system when interfaces are not defined and adhered to, but can succeed with the application of careful engineering practices. The second possibility involves participants that present a *heterogeneous* view of the domain and/or have no common objective. To connect systems that lack commonalities, a *system federation* is required and a method for the creation of the model that enables this federation is the focus of the Object-Oriented Method for Interoperability. [You02]

For the Department of Defense, system federations are the appropriate interoperability solution. Foremost, the legacy of DoD's acquisition structure is a mass of systems designed and built to last for decades. Legacy systems will therefore always be in use at any specific point in time, which introduces a prohibitive barrier to using a systems integration approach to interoperability. To develop an integrated system that includes legacy systems one of two possibilities exist. The new or non-legacy systems undergoing development could adopt the same legacy view and legacy objectives as the legacy system, thus preserving a homogenous view of the domain, or else the legacy system must be upgraded to the new, non-legacy domain view. Maintaining the legacy view limits the potential of new technology, while replacing or upgrading all existing systems each time the domain view changes is financially prohibitive and unnecessary. The assumption that legacy systems are included alongside new systems in the interoperability effort is valid and thus restricts the solution to one such as the development of a federation of systems.

Figure II-1 depicts a possible system federation involving a forward observer reporting, via electronic means, the position of hostile armor to an intelligence cell, which further distributes the information to an at-sea task force.

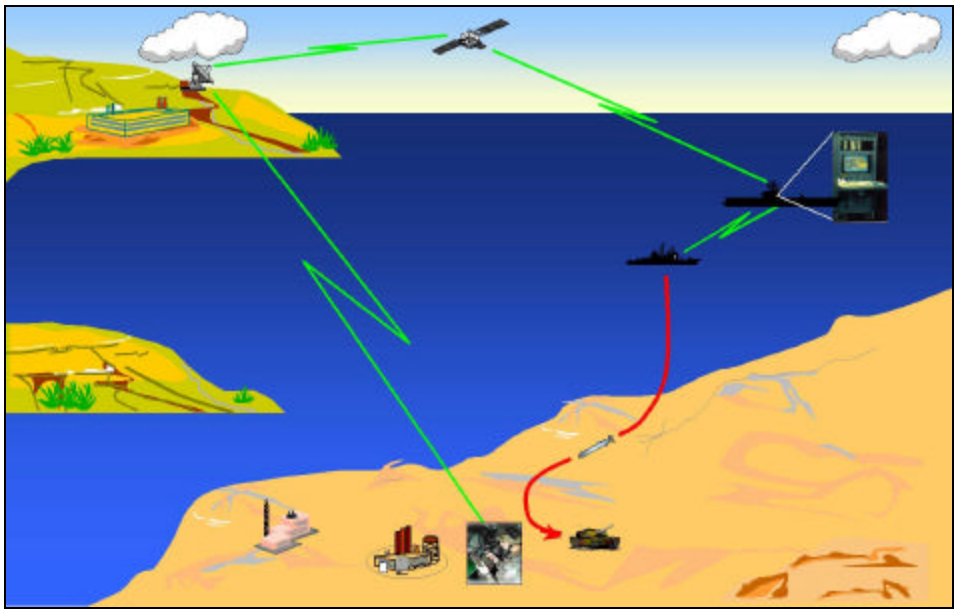

Figure II-1. Quest For System Interoperability. [from You02]

The task force plans and orders a task force asset to conduct a strike mission against the hostile and the cycle closes with the forward observer reporting that the strike mission destroyed the target. [You02] The prohibitive complexity of this simple example is the underlying mix of new and legacy systems that must be made interoperable in order to make the direct communication suggested by the example possible.

1. Interoperability Defined

> System interoperability involves not only the ability of systems to exchange information but also includes the capability for interaction and joint execution of tasks. [You02]

Joint task execution and information exchange are the functional capabilities around which a system federation is created. In Figure II-2 the impediments to the exchange of data are illustrated by exposing the system specific representations of the same real-world entity, the hostile tank. Each of the three representations captures a slightly different view of the real-world entity. [You02]

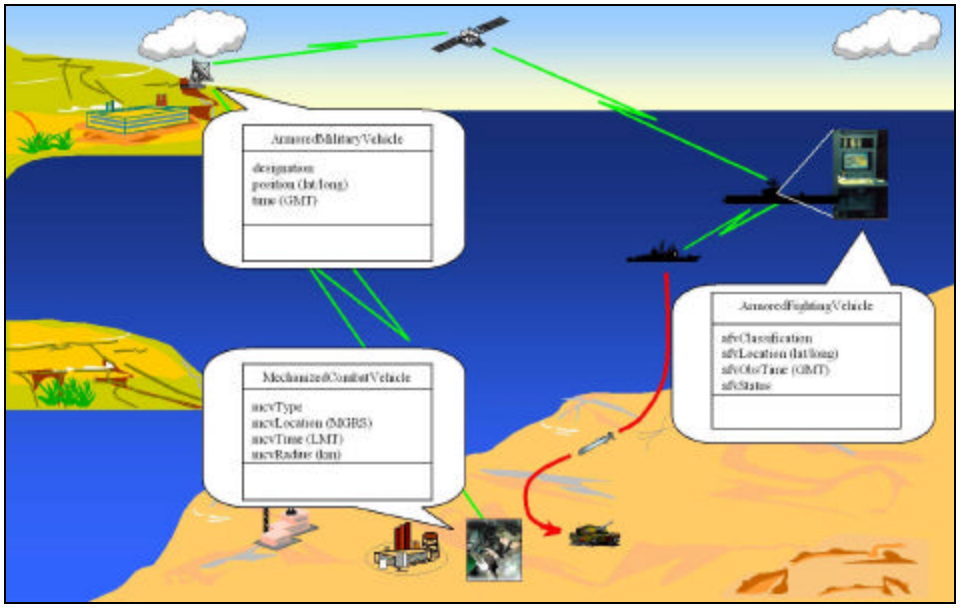

Figure II-2. Impediments to system interoperability. [from You02]

The forward observer's device represents the tank as a *MechanizedCombatVehicle* type that includes attributes for type of vehicle, location of the vehicle, time of report and maneuvering radius for the vehicle. The intelligence cell

represents the same tank as an *ArmoredMilitaryVehicle* type with attributes for description, position and time the tank was determined to be at the position indicated. Finally, the task force strike planning system represents the tank with an *ArmoredFightingVehicle* type containing attributes for classification, location, time of observation and status (i.e. operational, damaged or destroyed). [You02]

Young hypothesizes that a model-based approach facilitates the use of computer-aided resolution of data modeling differences, such as those introduced in the example above. The Object-Oriented Method for Interoperability (OOMI) defines the model that captures the data modeling differences of systems, provides the methodology for creating the model and describes the functional capabilities of a software application to aid in the construction and use of the model. Additionally, the construction of OOMI Translators -- for use at run-time to facilitate joint task execution and information exchange between the participants in the interoperable system federation -- are described. Figure II-3 shows the relationship between the major components of the OOMI.

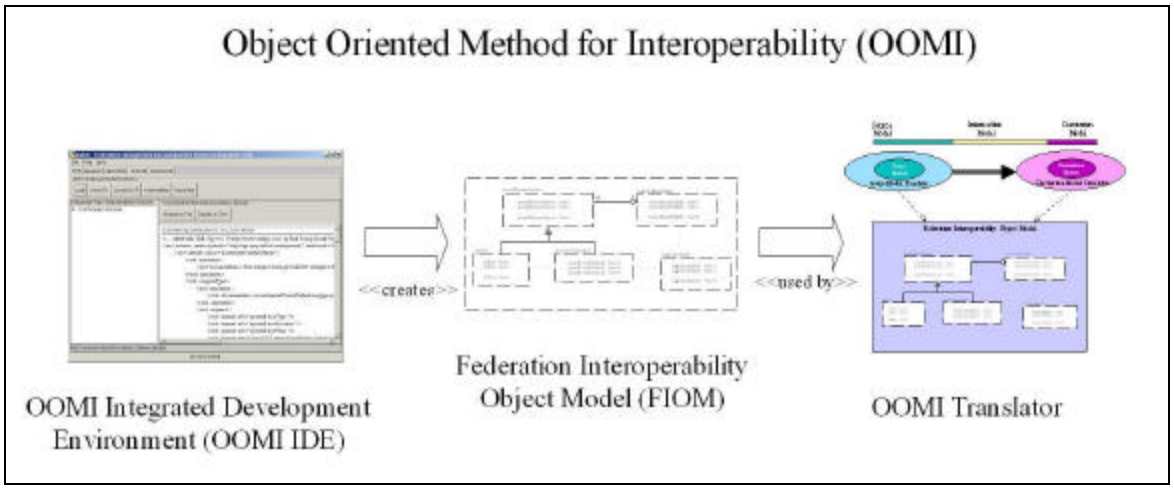

Figure II-3. Object-Oriented Method For Interoperability (OOMI) Key Components. [from You02]

The OOMI Integrated Development Environment (OOMI IDE) is the application and user level interface that provides computer aid for the construction and manipulation of the Federation Interoperability Object Model (FIOM). The FIOM is used to methodically capture, describe and resolve heterogeneity of representation and view of real world entities. Finally, an OOMI Translator accesses the Translations and

inheritance hierarchy of the FIOM to provide runtime resolution of differences in representation and view, enabling system interoperability. [You02]

2. Federation Interoperability Object Model

To capture and model differences in system representation of real-world entities, the set of possible differences, or heterogeneities must be decided upon. The OOMI consolidates a list of eight heterogeneities, which are:

Heterogeneity of Hardware and Operating Systems
Heterogeneity of Organizational Models
Heterogeneity of Structure
Heterogeneity of Presentation
Heterogeneity of Meaning
Heterogeneity of Scope
Heterogeneity of Level of Abstraction
Heterogeneity of Temporal Validity

These heterogeneities are further partitioned into two categories with respect to the real world entity being modeled. The first category contains heterogeneities of scope, level of abstraction and temporal validity, as these describe *what* information in particular concerning a real world entity the model captures. The second category contains the rest of the types of heterogeneity, which pertain to *how* the modeled information that describes the real world entity is represented. Within the FIOM, the *what* category is termed a *view* of the entity, and *how* category is termed *representation*. [You02]

> When two systems use the same set of features to model the same real world entity, they have the same view of the real world entity. Two systems that have the same view of a real world entity can nevertheless model the same feature differently. In this case, the two systems are said to provide different *representations* of the modeled feature. [You02]

Component systems are those systems under consideration for participation in an interoperability effort. Under the OOMI, component systems are registered in the FIOM in order to define the desired system federation. Component systems model real world entities and so an analogous concept, termed Federation Entity (FE), exists within the FIOM to serve as the standard federation model of entities being modeled. For each FE there can exist multiple real-world views that must be modeled by the federation. Each FE therefore contains one or more Federation Entity Views (FEVs), each of which

corresponds to a unique component system view of a real-world entity. The component system view and representation of a real-world entity is modeled completely by the Component Class Representation (CCR). [You02]

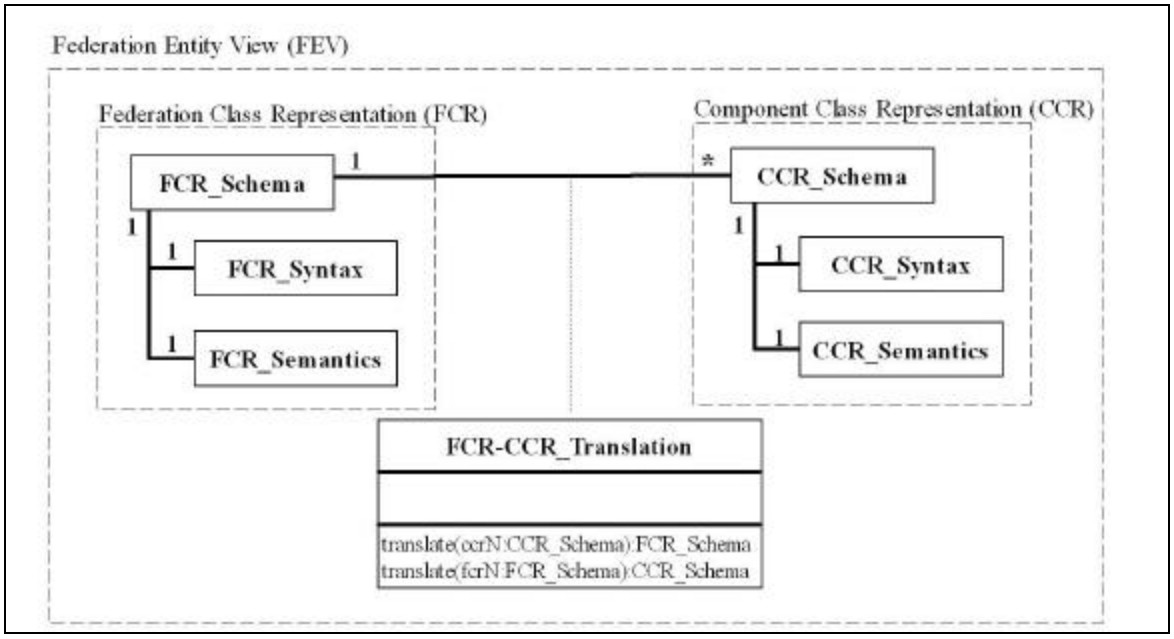

Figure II-4. FCR-CCR Translation Class. [from You02]

The CCR consists of a Schema class, Syntax file and Semantics files. By definition, each CCR relates to a single Federation Class Representation (FCR), which also contains a Schema class, Syntax file and Semantics file. The FCR Schema class models the specific FIOM representation of the Federation Entity that corresponds to the real-world-entity representation modeled by the CCR Schema class. To resolve differences in representation, each CCR Schema class is associated with its corresponding FCR Schema class via a Translation class. [You02]

As depicted in Figure II-4, each FCR Schema class can be associated with zero or more CCR Schema classes. Notice also that the association by way of Translation occurs entirely contained within an FEV. This means that the FCR and CCR share a common view of the real world entity and that the attributes and operations of a CCR Schema class must therefore be in one-to-one correspondence with the attributes and operations of the FCR Schema class. The FCR and CCR Syntactic and Semantic components are used by the Component Model Correlator to identify appropriate and corresponding FCR-CCR

pairs that share the same view of a real world entity. When the Correlation process does not find an FCR Schema class within the FIOM that is in one to one correspondence with a CCR Schema class, the difference in view necessitates the generation of a new FCR. [You02]

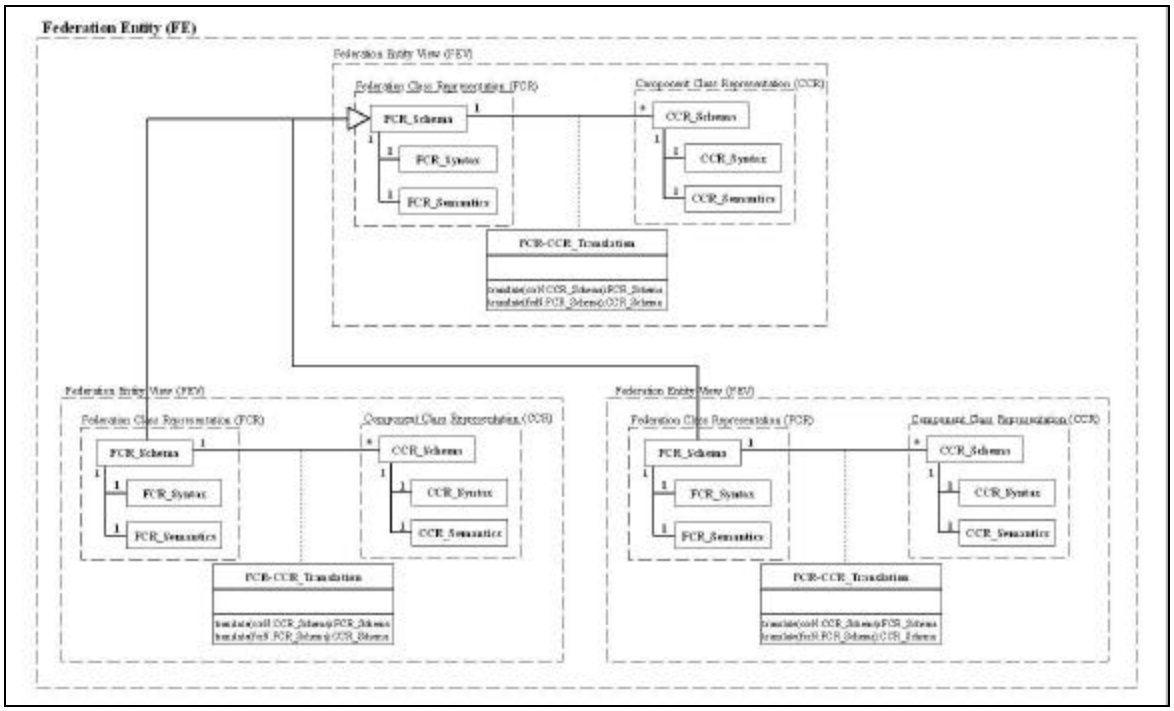

Figure II-5. OOMI Federation Entity (FE) Archetype. [from You02]

By generalizing or specializing an existing FCR Schema class, a new view of the Federation Entity is created. A view that is associated with the other FEVs via an inheritance relationship between FCR Schema classes. The resulting FCR inheritance hierarchy is thereby responsible for resolving modeling differences in view. Figure II-5 shows an example of three views of the same Federation Entity, associated via an inheritance relationship between the FCR Schema classes. By virtue of the substitutability properties of object inheritance, as described by Liskov and Wing [LW94, WO00], a sub type may be used interchangeably with a super type, resulting in the desirable ability to substitute one FEV for another, within an FEV inheritance hierarchy. [You02]

The modeling characteristics of the systems in the example from Figure II-2 demonstrate the necessity of the capability to resolve differences in both view and

representation. The FCR inheritance hierarchy for the example is shown in Figure II-6. An FE named groundCombatVehicle is created within the FIOM to model the real world entity (tank) from the example. This FE contains three FEVs and thus provides three distinct views of the tank. The most general view contains an FCR Schema class with three attributes that are in one-to-one correspondence with the attributes of the intelligence cell view of the tank. Given an intelligence cell CCR Schema class, a translation class resolving differences in representation can associate each of the attributes of the CCR Schema class with the FCR Schema class attributes contained within *groundCombatVehicle_View2_FCR.* The result of this process is that the CCR Schema class for the intelligence cell system is registered in the FIOM. [You02]

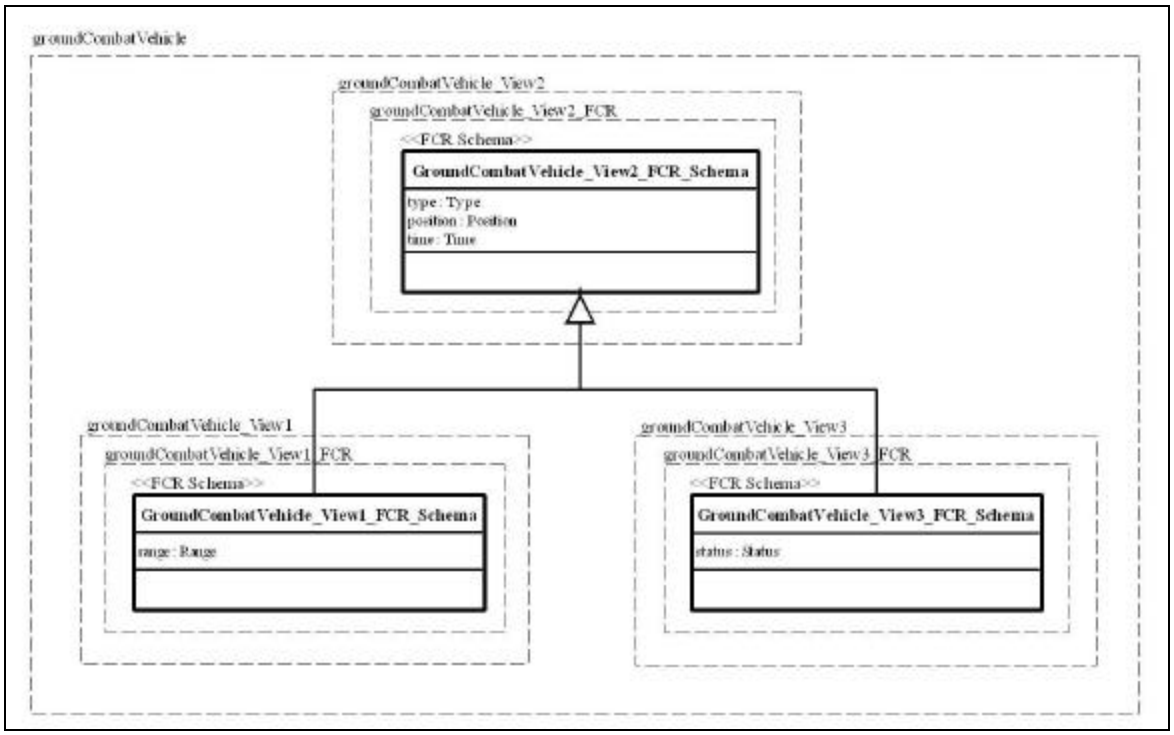

Figure II-6. Example FCR Schema Inheritance Hierarchy. [after You02]

The task force strike planning system models the tank from a different perspective and so, the *GroundCombatVehicle_View2_FCR_Schema* class is specialized to provide an additional attribute of type *Status* within a new class named *GroundCombatVehicle_View3_FCR_Schema*. An FCR Schema class, contained within FEV *groundCombatVehicle_View3* is now available and in one-to-one correspondence with the CCR Schema class representing the task force strike planning view of the tank.

Notice as well that a third view is achieved by again specializing the *GroundCombatVehicle_View2_FCR_Schema* class to provide an attribute of type *Range* within a new class named *GroundCombatVehicle_View1_FCR_Schema*. This view is in one to one correspondence with the model used by the ground observer's hand-held device. The differences in view of all three systems from our example are therefore represented within the FIOM, and can be resolved by simple substitution within the defined inheritance hierarchy. [You02]

3. OOMI Translator

At runtime, the focus returns to information exchange and joint task execution. The pre-runtime OOMI construction activity has produced the model from which interoperability is achieved between the component systems. An OOMI Translator uses the FIOM Translation classes and FEV inheritance hierarchy to resolve the differences in representation and view between component systems. System information intended to be exchanged is modeled and serialized in an Extensible Markup Language (XML) format to facilitate automated conversion of the XML-ized data into Java objects through a process known as data binding.

XML provides a scalable, open and extensible way to provide meaningful metadata within human readable file based storage. An XML Schema allows the metadata to be defined external to the storage, so that a single XML Schema can be re-used by multiple XML files that share similar markup requirements, and yet each XML file can be unique. For the OOMI, an XML Schema is created to represent the interface of each component system to be registered in the FIOM. A CCR Schema comes from an XML Schema that contains the definition of a component system's syntax and semantics for data modeling at the system's external interface. [You02]

Java classes are easily created from XML Schema documents using a data binding framework such as the Castor open source project.[Chr01] The value of Castor data binding is the addition of marshal and unmarshal methods to the resulting Java class. The unmarshal method provides the functional ability to instantiate a specific class instance from an XML document that conforms to the XML schema from which the class was created. Marshalling works in the opposite direction to create an XML document from

an instance of a class object, again with conformance to the original XML Schema. Figure II-7 shows the relationships between the Source XML Schema, CCR Schema, Source XML Document and CCR Schema Object. The process begins in the lower-left corner of the figure.

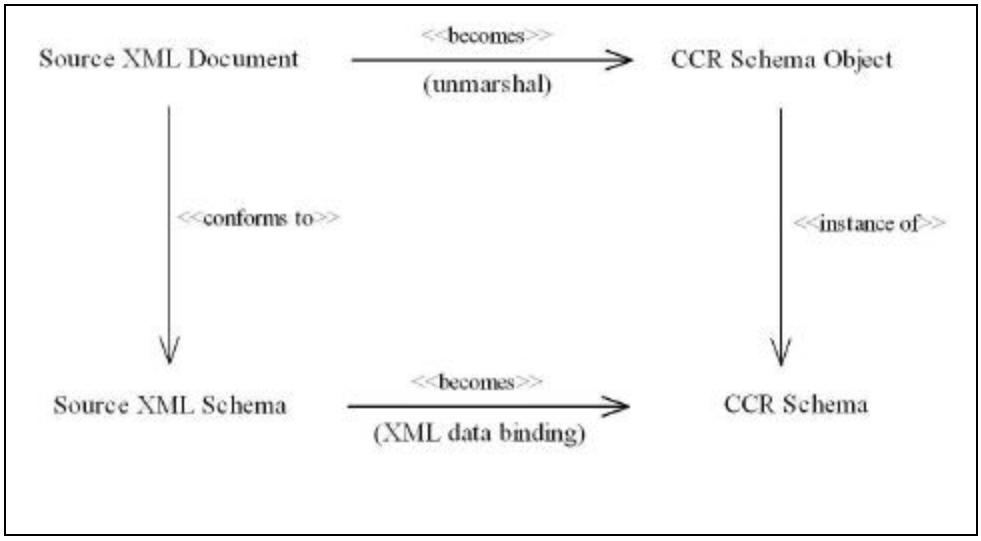

Figure II-7. Process For Converting Source XML Instance Document To Its Equivalent CCR Schema Object. [from You02]

At runtime, information exported from a specified system, as well as any external tasking request or response, are formatted as the Source XML Document, which is said to conform to the Source XML Schema. Conformance of an XML Document to an XML Schema can be checked through a process known as *validation*. A *valid* XML Document is *unmarshaled* by the unmarshal method of the CCR Schema class to create a specific instance of the class, a CCR Schema Object. The instance Object may also be *marshaled* back to an XML document, by the marshal method of the CCR Schema class.

During the process of registering a CCR in the FIOM, a Translation class is created that resolves differences in representation between the CCR Schema and FCR Schema. The Translation class converts a CCR Schema Object into an FCR Schema Object as shown in Figure II-8.

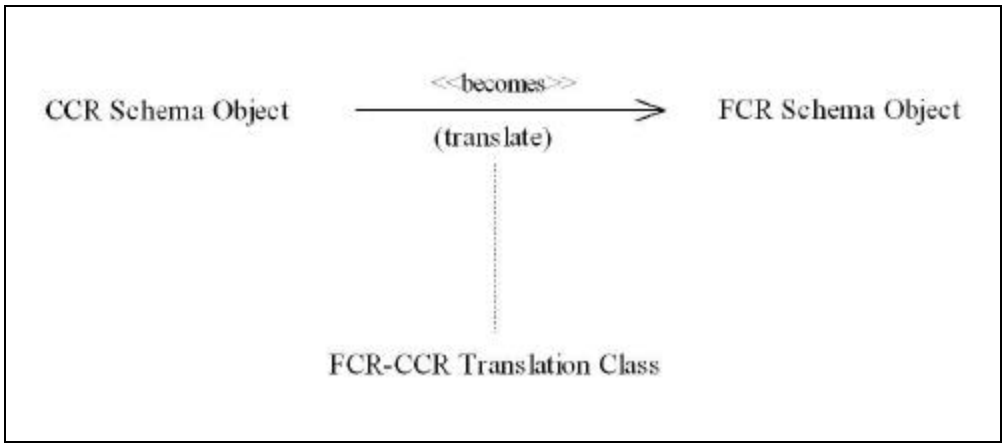

Figure II-8. CCR Schema Object To FCR Schema Object Translation. [from You02]

The OOMI Translator logically connects source and destination systems involved in information exchange or joint task execution. The component system acting as the source for the communication generates an instance of an XML document that conforms to its FIOM registered CCR Schema. The Source Model Translator accesses the FIOM and uses the CCR Schema unmarshal method to instantiate a CCR Schema Object from the source XML document. The CCR Schema Object is next translated into an FCR Schema Object by again accessing the FIOM and using the registered FCR-CCR Translation class. If the Destination Model holds a different view of the real-world-entity, the OOMI Translator will access the FIOM and traverse the FCR Schema inheritance hierarchy to translate the FCR Schema Object from the Federation Entity View corresponding to the Source Model into the FEV corresponding to the Destination model. Once the difference in view is resolved, the resulting FCR Schema Object is translated into the Destination CCR Schema Object using the registered FCR-CCR Translation class for the Destination CCR. Finally, the CCR Schema Object is marshaled by the OOMI Translator into a valid XML document that conforms to the CCR Schema for the Destination Component external interface.

One possible OOMI Translator architecture is presented in Figure II-9, in which each component system is logically wrapped in an OOMI Translator, referred to respectively as *Source Model Translator* and *Destination Model Translator* for clarity. In this particular case, the communication takes place in terms of the intermediate

(FIOM) model of the real world entity. Other arrangements include a single, centrally located translator, and a single wrapper translator surrounding only one system or the other.

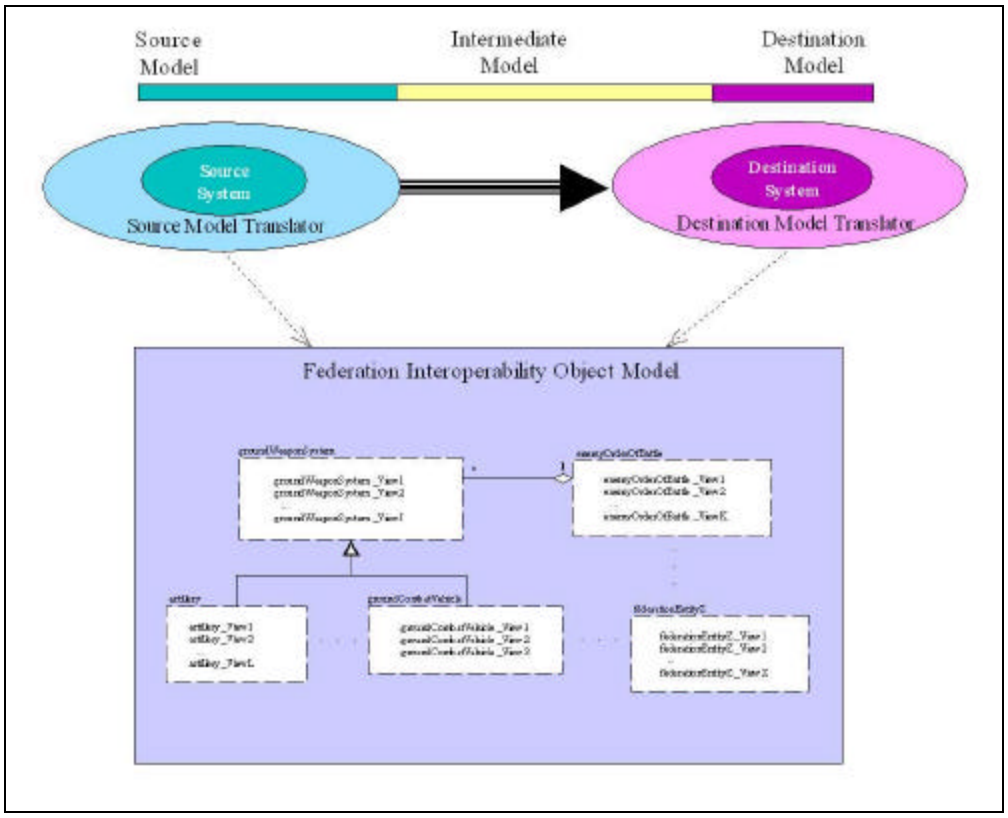

Figure II-9. Translator – FIOM Interaction. [from You02]

The OOMI Translator provides the logical link between component systems, forming the desired system federation. Unfortunately, the runtime environment is considerably more complex than the logical view of the OOMI Translator functional operation. The system federation has great potential to be physically dispersed and it is unrealistic to assume the existence of dedicated direct communication links between the component systems. The hypothetical battlefield environment suggested in the example from this chapter demonstrates the type of distribution and communication requirements common to the DoD environment. The FIOM and OOMI Translator may very well require a distributed solution to function in the target environment. The non-battlefield environment of DoD Acquisition is equally dispersed and includes participants from

multiple government and contract sources. The collaborative effort required to construct an FIOM may also require a distributed solution.

B. CHALLENGES OF DISTRIBUTED SYSTEMS

All distributed systems share a common set of challenges that are well known and for which acceptable solutions exist. In constructing a distributed system, however, complexity of the overall solution tends to increase when choosing a locally optimal solution for each challenge. Specialization in the distributed system allows complexity to be traded for better performance in a limited domain. Understanding the characteristics of the challenges is therefore important in the design of every distributed system. A list of seven problem areas make up the set of challenges which are commonly referred to by the terms Heterogeneity, Openness, Security, Scalability, Failure handling, Concurrency and Transparency.

1. Heterogeneity

Heterogeneity happens in distributed systems when a solution involves the use of several differing types of hardware and software. The components of the system may or may not be designed to interoperate directly and in some cases may be completely incompatible. Middleware is applied to support the communication and cooperation between incompatible components to allow creation of the distributed system. In this solution the middleware hides the details of the interaction of components and presents a common central view of the interaction. For example, CORBA (Common Object Request Broker Architecture) provides this common view for interaction between the objects of a distributed system by using IDL (Interface Description Language) to document and advertise available distributed resources via their CORBA compliant interface. For programming languages there exists a similar concern that all code should be executable on all nodes of a distributed system. The common middleware approach to this problem involves a virtual machine that runs on each host and provides a common environment for code to execute, on top of the underlying heterogeneity. There are many examples of solutions involving middleware. The primary concern is not which solution is chosen, but rather the effect the choice has on the complexity of the system and the restrictions that are placed on the available solutions for the remaining challenges. In one

sense, a middleware solution may reduce complexity by defining a standard, but this solution may limit Openness, as is the case with proprietary standards. Another middleware solution may adversely affect Scalability, Security and Transparency. In every case, the solution will involve some method of abstracting the heterogeneity, and the effect of the implementation on the other challenges must be considered. [CDK01]

2. Openness

Distributed systems are developed to promote sharing of resources. A system exhibits openness when existing resources are available to new services or methods of accessing the shared environment. The two common approaches to ensuring openness involve either publishing the specification of the interfaces and components, or implementing based on a standard. Published implementations are limited in extension and reuse by the ability to understand the original intent and methodology. Openly available documentation published in a standard format is one solution. The Request For Comment (RFC) process of documenting networking technologies aided the development of the Internet during the 1980s. [CDK01]

Implementation based on published standards is the second method of creating a distributed system open to extension. In this case, deciding on the form, function and purpose of the standard is the first challenge. International Standard Organization (ISO) and World Wide Web Consortium (W3C) are two examples of international efforts to promote openness in current and future computing technology. Producing and maintaining a standard is not an easy process, particularly when approached in a top down fashion. The overwhelming reality has been an existing technology, which based on functional popularity and consumer demand becomes widely implemented and thus, emerges as a de facto standard long before an organized effort to standardize. The ability of a technology to rapidly spread in this manor is directly associated with its degree of openness as many independent competitive efforts combine to produce the phenomenon.

Most recently, the focus has shifted to embedding the documentation in the code as well as in the data. Resource Description Framework (RDF) for the Web and Javadoc for the Java programming language are two examples of current efforts to embed meaningful description in order to promote openness. RDF embeds semantic meaning

into Web content, an example of marking up data for extension and reuse in an open environment. Javadoc provides an automated method of providing documentation for Java classes to promote reuse and collaboration in support of the open source movement.

3. Security

For distributed systems, the security challenge is made up of three components. Integrity and confidentiality are common security concerns for many systems, but the third component, availability, increases the complexity of the security solution for the distributed environment. In most cases the network connectivity supporting the system is not controlled by the entity operating the distributed system. While encryption techniques safeguard the content passing over the intermediate links, the availability of the network cannot currently be guaranteed. Solutions that improve availability, such as redundancy or dedicated infrastructure, limit scalability by increasing cost and or complexity.

Trusting the integrity of executable code received from external sources via the network remains a concern, despite the existence of techniques for marking and signing digital material. The basis of trust issues for distributed systems exist not because of paranoia between collaborators, but due to the continuing inherent insecurity of the operating systems in use by the host nodes throughout the network. While the communication between nodes can be effectively safe guarded with encryption, the host systems themselves remain vulnerable to exploitation. There exists no complete solution to this challenge and so carefully managed risk and complicated processes dominate the solution to the security challenge. Providing assurance of security in a distributed system many time limits the degree of heterogeneity, the scalability or the transparency of the solution.

4. Scalability

Scalability of a distributed system implies that the system is effective despite increases in available resources or users. The challenges of designing for scalability include controlling cost, maintaining performance and maintaining adequate supply of resources. Each of these challenges is directly related to the others. For example, changes to a distributed system that positively affect performance are attributed to an

increase in a particular system resource, which is realized by incurring an increase in cost. When a system is designed to scale up, adding resources will in the worst case maintain current system performance and can be achieved in such a fashion that cost is not prohibitive. In most cases, financially affordable growth translates to adding nodes to a distributed system without replacing or removing existing participating nodes. [CDK01]

Date	Computers	Web Servers
1979, Dec	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866

Table 1. Computers In The Internet. [from CDK01]

The Internet is the premier example of a scalable distributed system, the growth of which is shown in Table 1 and the success of which is commonly known. Specific scalability challenges that were overcome to facilitate the expansion of the Internet include distribution of performance requirements, technological advances that reduced cost, and shared rather then monolithic construction and management of infrastructure. [CDK01]

5. Failure Handling

Planning for the inevitable failure of some portion of a distributed system takes the form of one or more of five common handling techniques: detection, masking, tolerating, recovery, and redundancy. Detecting failure is of value to any approach but is not feasible in every situation. Determining the existence of a failure on a remote system, for example, is complicated by the possibility of intermediate failure and the desire to maintain low administrative overhead. When failure is detected, re-performing the failed action or re-directing to a duplicate resource may mask it. In some cases failure is accepted and tolerated without corrective action on the part of the distributed system. Recovery as a handling technique restores a previous state of the system after an event

such as a crash causes corruption to data or programs. Finally redundancy hopes to guarantee alternatives while accepting the reality of failure. [CDK01]

Numerous useful solutions exist that mitigate this challenge; however, considering the effect of the chosen solution on the other challenges is important. Tolerating failure may lead to lack of availability, thus countering a primary security concern. The ability to detect failure increases complexity and may impact Scalability. The purpose of the system and corresponding requirements determine the degree and method of failure handling and must include acknowledgement of the risks associated with the effects on other challenges.

6. Concurrency

Sharing in any context introduces the possibility of multiple attempts to access a single resource. The effect of concurrent access is lack of assurance of the consistency of the results. Order of access is important in many contexts and if order is not maintained during concurrent access, the final outcome will be incorrect. The classic semaphore solution from the operating system domain is useful in the distributed environment to address this challenge. The primary concern involves the re-use of software components not originally intended for distribution and not designed to support concurrency. Action must be taken to ensure consistency of data and testing conducted to verify the existence of safe access processes. [CDK01]

7. Transparency

Transparency in distributed systems can be thought of as the ultimate in abstraction and is the most important challenge. This challenge can be considered at many levels, the highest of which is the general idea that for a distributed system to be considered useful, the user must not be able to tell what resources the distributed system is using to process the user's requests. Similarly, Application Programmers should be provided high-level programming abstractions and interfaces that hide the details of the distributed system's low-level complexities. The transparency challenge is composed of several specific sub-topics, commonly known as transparency of access, location, concurrency, replication, failure, mobility, performance and scale. [CDK01]

Achieving access transparency requires abstracting the existence of local and remote resources such that both are accessed via identical means. The concept of mounting a remote file share, as a branch of the local directory structure, is one example of providing access transparency. Without this transparency, a determination between local and remote must first be made, followed by choosing the appropriate access method for the type of resource. Closely related to this is location transparency, which abstracts the absolute physical location of resources so that they may be referenced in the same terms as resources native to the host platform. Transparency with respect to concurrency, replication and performance simply means that the processes allowing these functions are abstracted by the system. Shared resources may be accessed concurrently, such as a database, or may be handled with multiple replicated sources, as demonstrated by the Web. The configuration of shared resources are linked to performance of the system and associated with handling variations in load. Failure transparency implies that functionality persists despite underlying failures, meaning that both the failure and the additional effort required to assure success of a system function are hidden. Finally, mobility transparency for a distributed system allows the action required to account for and resolve movement of the systems within the distributed environment. The recent expansion of networking into a user level wireless domain increases the occurrence of computing resource mobility. [CDK01]

C. COMMON ARCHITECTURAL STYLE

Every design effort is based on a high-level conception of the solution, often referred to as an architecture. The type and number of components, the intended outwardly visible functional behavior and the method in which the components fit together to form the solution are described by the architecture. Over time, certain arrangements prove useful in many problem domains and thus emerge as recognizable styles of architecture. Terms like *Object-Oriented* and *Layered* communicate the majority of the properties of the final solution, despite the fact that each software system possesses uniqueness of implementation. This section introduces several of the most common architectural styles and presents a high-level description of primary characteristics of each.

1. Software Architectures

Data-centered, data-flow, call-and-return, object-oriented and layered are the five prominent architectural styles that are used by the majority of software design efforts.

A common view of a computer system involves a machine that helps a person either access or update a central data bank of useful information. This Data-centered style is natural, because it approximates the classic human solution, in which learned information is recorded and stored for future access. Cataloged libraries are the physical domain analogy to this architectural style. Data consumers retrieve specific information from the central storage and do something with the information, perhaps adding, subtracting or changing the contents of the data store in the process. A monolithic database is the most commonly thought of application that implements this style. [Pre01]

Data-flow is the style that represents the automation of a process. In this style, data enters a software program as input in one form and exits as output, perhaps in a different form, with the assumption that some transformation of the data may occur in the interim. In relation to the data-centered style, data-flow represents the processes a data consumer may apply to data retrieved from a central storage. [Pre01]

Computer programming combines data-centered and data-flow to provide useful automation of complex mathematical or engineering tasks. Call-and-return improves programs by allowing greater flexibility in progressing toward the goal. One portion of a software program calls to another to conduct a calculation or perform its unique function. Upon completion the result is returned to the caller. The development of this style leads to parallel processing, modular design of programs, and distribution of processing. For distribution, the call-and-return mechanisms are represented as communications between remote sites. [Pre01]

The object-oriented style of architecture depends on information hiding and functional abstraction. In the object model, the data is encapsulated by the operations that are allowed to be performed. Within an object, the details of the operations are hidden and direct modification of the data set is prohibited. This style promotes

maximum re-use of programming components and enables a purely modular approach to constructing large software programs. [Pre01]

A layered approach to software produces concentric abstractions that build upon each other to hide complexity while providing higher level functionality. In purest form each layer completely encapsulates the sub components, eliminating direct interaction and replacing it with services provided by the layer. Boundaries between layers should be distinct and maintained by interfaces. The interface model allows a description of the possible operations available to be defined and held constant, while the details of the underlying implementation may change. When interfaces between layers are maintained, specific instances of entire layers can be removed and replaced without requiring changes in adjacent layers. [Pre01]

A distributed system's characteristics are determined by its software style, however the inclusion of multiple software components, multiple hardware components and the propensity for physical dispersion introduces more complex architectural concerns.

2. Distributed System Architectures

For distributed systems the functional responsibilities of the whole are divided between the sub-components. Several architectural possibilities exist for distributed systems, though all provide essentially the same function, description of the delegation of responsibilities within the system. Foremost of the architectures is the client-server model in its various forms. Other models include *services*, *proxy* and *peer-to-peer*.

Client-server involves a division of labor primarily intended to support a one-to-many relationship between the data or processing and the consumers. The server portion is primarily oriented according to the data-centered or data-flow software styles. A large amount of resources or exceptional capability is centrally located as a server to provide for many distributed consumers, termed clients. The client and server side connect by way of a messaging protocol very similar to a call and return mechanism. [CDK01]

Mobile code is a variation of client-server in which executable code transfers from the server to the client for processing, rather than executing on the server.

Originally intended to reduce server side processing on behalf of client demands, the more useful capability is the distribution of client-side applications on-demand. Mobile code led to the advent of *mobile agents*, which conceptually reverses the responsibilities. A mobile agent usually originates at a client and executes on a server, returning the result to the client. There are considerable security challenges associated with both Mobile code and agents because of the lack of ability to provide assurance that executable code can be trusted not to violate integrity or privacy on the host platform. [CDK01]

Networked computers conceptually receive all executable code from a central server, primarily to reduce the cost associated with maintaining many distributed clients. In this model, the client is an entire computer, rather than a single application or process running on a host. The client boot process downloads operating system, applications and data. Taking this concept a step farther, Thin-clients are Networked computers that provide the minimal amount of local support, typically just the graphical user interface application, while providing for all the data storage and processing on a server. Various Thin-client implementations divide the display responsibilities at different levels. At one extreme, the user interface is maintained on the server and screen images in terms of pixels are sent for display on the Thin-client. At the other extreme, such as the Web's HTML, only the data is transferred and the Thin-client is responsible for all display functions. [CDK01]

The server portion of a simple client server architecture is one element that influences the upper bound on the maximum number of clients. When a client-server system must scale beyond this bound, a more complex architecture of *services* is one possible solution. The services model uses multiple servers and replicated data to increase the number of data sources available to the client pool. Replication between the servers providing the service implies that each individual server maintains a complete set of the data. This requirement creates significant overhead when the size of the data set or number of required copies of the data increases. [CDK01]

A less complex approximation of the services model is the proxy model. In this case, proxy severs separate clients from servers or services. The intermediary presence of the proxy hides the number of actual clients from the server, thus reducing the number

of connections to the server or service. The proxy also caches the most recent data, which potentially decreases the number of client requests to the server. If more than one client requests the same data, the second and subsequent requests are satisfied by the proxy, from data stored in that cache. This architecture eases some of the scalability issues with the client-server model, with less complexity of the services model. The down side of proxy servers is that cache is a finite resource and old data must be overwritten constantly when the cache is full, despite the fact that the very next client request may be for the data to be overwritten. There is no way to tell which data will be requested in the future, and so caching has limited applicability. [CDK01]

Peer processing flattens the hierarchical structure of client-server based approaches and distributes the data and processing responsibilities as equally as possible between the peers. The benefits of the peer model are better upward scalability and less complicated distribution of shared data. Client-server is the less complicated approximation of peer-to-peer processing. Peer processing is an important capability for future distributed systems and its architectural style continues to evolve.

A currently popular model for implementing peer networking involves the use of an application layer software overlay that approximates the peer processing architecture. The term overlay describes the fact that the peer networking abstraction hides the current Internet infrastructure. Logically, peer interactions appear to be direct collaboration and even multicast, but without infrastructure level support for multipoint peer sessions and multicast delivery of data packets the true communication mechanisms remain tied to the client-server model at the lowest levels. Client-server architectures overlay the existing networking infrastructure with a system of point-to-point connections originating at the client and ending at a central server, or service. Overlaying a peer architecture does the same thing, thereby preserving the ability to use point-to-point connections to allow collaboration between peers, as well as providing multicast for distribution and multi-source for retrieval. [BCM+02]

Peer overlay networks provide multicast without requiring changes to the network infrastructure by using processing resources available to the peers at the application layer to relay messages sent to a group via multicast. Controlling the density and distribution

of a peer network is important in order to maintain performance by sharing the processing load. One solution produces an artificial hierarchy by dividing the peers according to sub networks so that a hierarchy of clusters exists. In this way multicast distribution via the overlay is expedited by delivery via the hierarchy to a single peer representing the entire cluster. This peer then disseminates the message to the rest of the cluster. Because the solution chooses the clusters dynamically, the logical arrangement tries to keep clusters nearly equal in size and attempts to compose clusters of peers that are *close* to each other in terms of packet propagation delay across the existing Internet. [BBK02]

In reality, part of several architectural styles combine to form a design architecture for a specific application. In choosing the style components to combine, the entire problem domain must be considered to determine the impact of the specific domain characteristics on the future applicability of the style choice. Complexity increases as more types of architecture are combined, but the functional capability of the resulting distributed system may be dramatically improved. The propensity for any particular system to be made up of many sub components from differing architectural sources makes direct comparison of distributed system implementations problematic.

D. DISTRIBUTED SYSTEMS CLASSIFICATION PARADIGMS

The ability to compare and contrast distributed systems is important when contemplating a distributed solution to a problem. The existence of the common challenges of distributed systems and the set of architectural components from which distributed systems are constructed suggest several predominant paradigms that provide a view of the possible solutions domain, against which distributed systems can be classified and thus compared. Research in the area of distributed systems has produced prototype systems and fundamental descriptions of the techniques and approaches useful in creating such a system. Eight such research efforts, 2K, Globus, Legion, Sombrero, StratOSphere, Project Oxygen, Opus, Globe and Distributed CAPS, have further been considered and classified according to the paradigm model. The paradigm comparison process requires a summary of each system that identifies the unique concepts. The concepts are then mapped against a set of paradigms and then the systems can be compared and contrasted to produce a relative view of the systems as they relate to each

other. The paradigm model itself is based on a partitioning of the general distributed systems solution space. Section II.D.1 describes the partitioning and Section II.D.2 presents some examples to demonstrate the classification results.

1. Conceptual Abstraction for the Classification Paradigms

There are several paradigms to consider within the purview of distributed systems. The paradigms exist because of the differing purposes for distributed systems. A distributed system is considered to be most effective at a particular type of task, and so the solution space for the creation of possible distributed systems is partitioned into *User Centric* systems, *Processing/Storage Centric* systems, *Implementation Centric* systems and the abstract parent of them all, *Generalized* systems. [ML02]

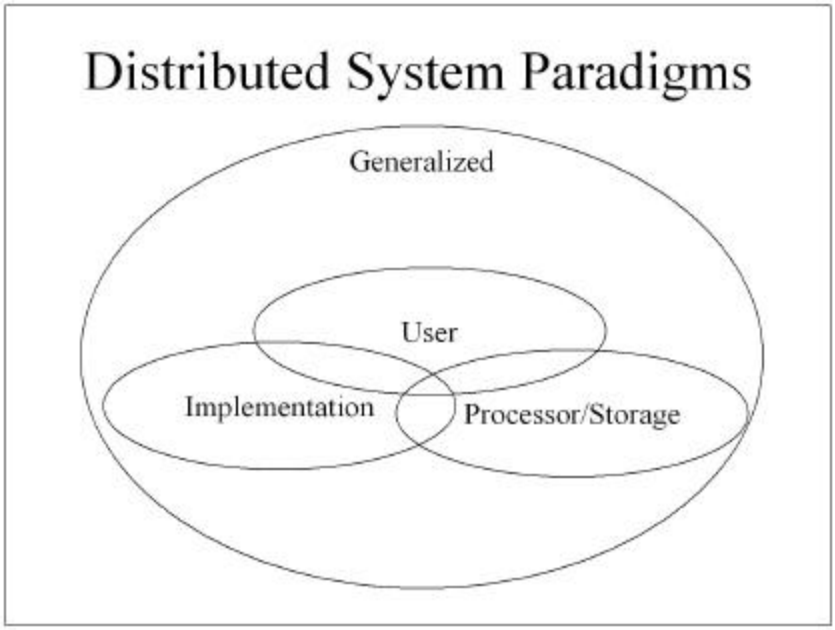

Figure II-10. Distributed System Classification Paradigms [from ML02].

User Centric focuses the characteristics of the system around user concerns, so that the system is very good at supporting distributed users. Processor/Storage Centric systems focus on building distributed systems that support massive parallel processing and massive data sets, often using existing networked systems. Implementation Centric distributed systems focus the effort on using implementation architectures that provide facilities explicitly designed to deal directly with the specific challenges of distributed

systems. This type of system could be used for any end application, making it the most generalized, but the real focus is on getting the most complete, truly distributed system, even if starting from scratch is required. The Generalized classification is an abstract catchall included to provide completeness of the paradigm model. Other paradigms may exist and could be added to the model when defined. [ML02]

In terms of functional behavior, all types can emulate the behavior of any other type, however the implementation details may be extremely complex and the efficiency of the emulation is likely to be less than that of the emulated type. For example, a Processor/Storage Centric approach could be used to emulate a User Centric model, but the overhead of doing so will be inefficient, as the Processor/Storage Centric methods would make it possible for any user to transparently use *every* node in the system, which is physically unlikely in a very large distributed system. The goal of User Centric methods is to allow any user to use *any* node, a somewhat less complex problem. A Generalized distributed system would do everything well, and is an abstraction that cannot be realized. Sub-paradigms by definition sacrifice generality in order to optimize for a specific implementation and in order to obtain a good approximation of the specific end goal of a distributed system. The classification mechanism provides a means to group and compare distributed systems based on the choices that are made in mitigating the common challenges.

In the first paradigm, User Centric, the resource requirements of the user are small, so that any node in the distributed system has most of the required resources. The user is allowed to roam and may access any node of the distributed system, however, so the cases in which the local node does not have sufficient resources must be handled. Access must be transparent to the user in such a way that the user is presented a recognizable environment, no matter which host in the distributed system provides access. 2K is the example system of this variety, in which primarily user data is migrated to support a user's environment where and when the user accesses some node in the distributed system. When a local node cannot provide all of the user's processing needs, a remote node may be used to conduct the processing, passing the end result to the user's local node. [ML02]

Second we have the Processor/Storage Centric view the examples of which, known as Globus and Legion, are commonly called Grid Systems. These systems transparently make significantly more resources available, for example providing access to multiple processors for distributed execution in parallel. In this type of distributed system, the goal is to distribute storage and/or processing, so that code and data are primarily what migrate across the system. In this type of distribution, users are assumed to be more or less stationary, with respect to their access point for the distributed system, but their needs are such that the local node cannot efficiently provide for storage and/or processing. Additional resources from across the grid are dynamically and transparently requested and used to process the user requests. Location of processing is transparent, but in most cases orders of magnitude more nodes are involved than the one or two nodes per user involved in the User Centric paradigm. [ML02]

The third view is the Implementation Centric view, in which the use of direct implementation methods for dealing with the challenges of distributed systems is the main distinguishing feature. This classification is considered to be the most general of the three sub paradigms because of its runtime goal of supporting a broad array of distributed functions, including User Centric and Processor/Storage Centric functions. Often this direct approach will not use an overlay and thus requires new hardware and/or a new operating system. The Sombrero Single Address Space OS is our first example of this type of solution and chooses new hardware to provide directly for a large address space. The other example system classified as primarily Implementation Centric is StratOSphere, which advances a framework for distributed objects and mobile code applications. StratOSphere's object-oriented Java code does not require a new underlying operating system, but does require several layers of abstraction. The complexity of these layers rivals Sombrero's need for a new operating system but avoids the new hardware requirement. For both systems the end result is the same, simplification of distribution by providing direct support for the specific requirements of distributed applications. This paradigm is the most complex and expensive to implement of the three. [ML02]

2. Examples of System Classifications and Justifications

When distributed systems are classified according to the paradigms the goal is to present a visual representation such as the example shown in Figure II-11. Ellipses representing the characteristics of a particular distributed system are superimposed on the paradigm diagram from Figure II-10. The underlying paradigm diagram ellipses establish regions representing containment within the Paradigms, as well as areas of relative overlap between Paradigms. A system ellipsis is placed with its center of mass contained within the paradigm ellipse most fitting the design goals of the system in question. The system ellipses are oriented and sized so that their edges cross paradigm boundaries as appropriate to indicate to what extent their classifications include aspects of the other paradigms. Classifications may be based on descriptions and summaries of the systems and are not necessarily derived from empirical observation of actual systems or code. [ML02]

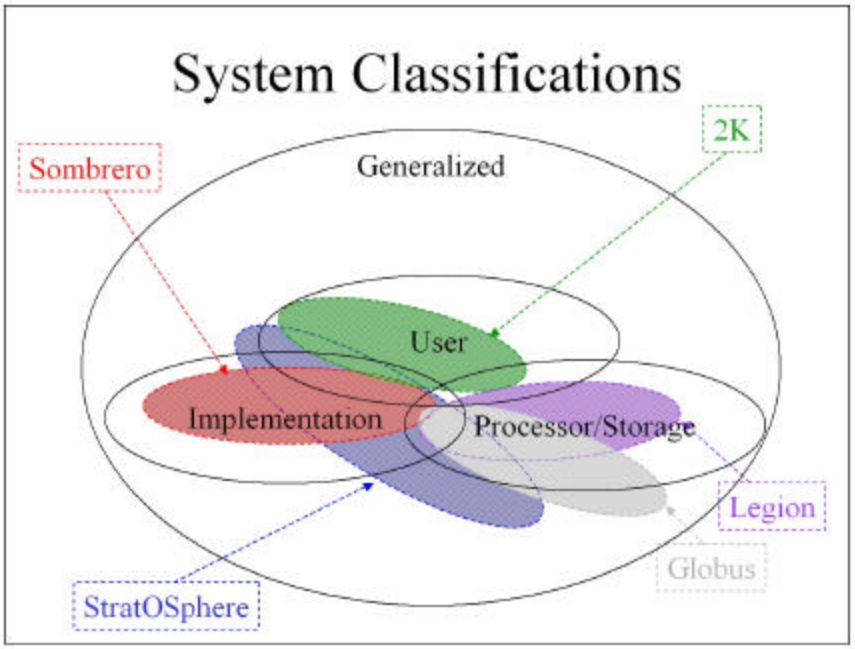

Figure II-11. Example Classifications Of Distributed Systems Research [from ML02].

The classifications for five of the example systems are depicted in Figure II-11. The classifications for the other three systems are shown in Figure II-12. The current

paradigm model uses the Heterogeneity, Scalability and Transparency challenges to frame the classifications. Five specific Transparencies, Access, Location, Concurrency, Mobility and Replication are included in the paradigm model. In the future, the model may use all of the common challenges and transparencies from Section II.B to provide more exacting classification. Considering only a subset of the challenges is sufficient for relative comparisons. [ML02]

Heterogeneity is assumed by all five systems, though each handles this challenge differently. The two general ways to deal with heterogeneity are to handle it directly and to abstract it away, hiding it behind a homogeneous layer. There are also ways to combine both approaches. In general, the direct methods for dealing with heterogeneity are considered Implementation Centric, while indirect methods are considered to belong to the other two paradigms. [ML02]

2K's Execution Environments allow for dynamic customization based on available resources at any host in the distributed system, despite heterogeneity of the hosts. This direct approach to heterogeneity is made possible by reflection, which is the process by which executing code can query the system. Users are represented in 2K by Execution Environments that are built out of components. Components carry with them certain resource requirements and cannot be used unless these requirements are met by the host system. 2K uses reflection to query host systems and determine the availability of required resources for a particular environment component. If any given component cannot be supported then it may be possible to provide the functionality of the component via a remote host, but there are cases when a component will not be available at the host in question. This best-effort approach is required because 2K advertises that users may roam to any device in the distributed system and obtain service. Although the service may be degraded, some level of service is always possible. This approach to heterogeneity is categorized as *combined* because reflection is a direct method but the 2K software overlay that must be installed on the participating host system is an indirect method. The purpose of this combined approach is to facilitate a user's seamless movement between host devices, thus supporting the predominately User Centric classification. However, the inclusion of reflection by 2K demonstrates the

Implementation Centric approach, and so the classification ellipse intersects that paradigm as well. [ML02]

Globus and Legion, on the other hand, establish a grid abstraction on top of the heterogeneity introduced by the host systems, thus indirectly presenting a homogeneous environment to the grid applications. Versions of the grid software must be pre-compiled for particular host environments, but the runtime environment provided by an operational grid is not aware of underlying heterogeneity. Globus and Legion both use a global naming scheme to provide addressing for the entire grid, which is considered Implementation Centric. This naming scheme involves indirection due to name resolution via naming services, which is an indirect solution, therefore limiting the extent of the Implementation Centric classification. Legion is considered to be less general than Globus because of the scoping of the project and the more focused application domains. Globus is considered more general, because its focus is research and the exploration of grid technology, while Legion focuses on specialty grid and application construction. [ML02]

To gain the benefit of a single namespace without the overhead of indirection, Sombrero uses 64-bit processors capable of providing hardware support for large address spaces. By using the large space to address an entire network and then routing by address (vice name), Sombrero achieves the same resource sharing capability of a grid system, at hardware speeds. Because Sombrero uses specialty hardware and requires an entirely new operating system, a prototype is not available and the implementation method in which heterogeneity is provided for remains an open issue. The assumption is that the direct approach would be used in this type of start-from-scratch distributed system, thus leading to an Implementation Centric classification. [ML02]

StratOSphere handles the heterogeneity challenge by implementing in Java, which uses virtual machines running on hosts to level the heterogeneous field into a single homogeneous runtime environment. Additional support for heterogeneity in the communications realm is provided by the layered architecture that includes support for many notable interoperability frameworks such as CORBA. For StratOSphere the

layered architecture provides direct support for many of the challenges of distributed systems, and so the Implementation Centric paradigm is fitting. [ML02]

Scalability as described previously involves striking a balance between cost, number of users and number of hosts. The Scalability challenge for 2K, Globus and Legion is handled directly by the system design so that on the surface it appears that scaling up is not an issue. For each of these example systems, however, there exist inherent assumptions that can cause bottlenecks to form as the systems are scaled up.

Consider that participation in all three systems is made possible via a user-level registration process. This requirement is accompanied by the need to install a host system specific software overlay. For all three projects, there may now or in the future be a specific type or version of hardware or operating system for which the distributed system overlay software is not compatible.

Beyond that problem, the next scaling issue is the performance bottleneck that will be experienced on the links between the nodes of the distributed systems. In this example, consider that Legion depends on the existing network infrastructure, including protocols, and therefore is limited by characteristics of the existing links between the systems. The concern being that at some point, the majority of the user pool could be separated from the majority of the processing resources such that most grid/user interaction must pass over a few or even only one physical link. As the number of users increases at a given location, the users may be isolated from the grid because of competition and congestion on the links between the users and the grid systems. In 2K we expect the users to be distributed, which helps avoid the bottleneck issue as more users are added. The suggested benefit of 2K, however, is support for mobile computing on any device including handheld devices. The wireless nature of these handheld devices introduces a bottleneck that limits the ability of the mobile device to participate in a distributed system equally with a wired device. The specific bottleneck concern is that migrating a user's 2K Execution Environment via a wireless link will require more time than the user intends to spend using the device.

A Scalability concern effecting grid systems in general is that when new hardware is added to the grid, it will likely outperform older hardware. Over time this performance gap will widen so that the relative delay associated with distributing processing cycles to older systems will cause the cost to be greater than the benefit of distribution, thereby introducing a load balancing problem. The concerns about the Scalability of 2K and Legion limit the extent with which the systems can be classified as Implementation Centric. Globus on the other hand is a more general grid system than Legion and so can be classified as being relatively more Implementation Centric. [ML02]

The ability for Sombrero to scale-up is unclear, but it is logical to assume that the upper limit of a 64-bit addressing scheme can and will be reached. Scaling beyond the 64-bit scheme requires an indirect solution and so the Sombrero classification extends into both the User and Processor/Storage Centric paradigms. The layered structure and object-oriented approach of StratOSphere is designed to directly support Scalability. The architecture of StratOSphere includes support for many existing distribution frameworks at its lowest layer. Despite the overall Implementation Centric classification, a specific instance of StratOSphere may be limited by using a remote invocation service. For this reason the classification extends into the User and Processor/Storage paradigms as well, meaning that Scalability may be handled either directly or indirectly depending on the StratOSphere Transport Layer invocation method and the purpose of distributed application in use. [ML02]

The classification of the systems also takes into consideration the affect of Transparencies. Each distributed system chooses to provide direct support for some subset of the Transparencies introduced in Section II.B.7 and must at the same time choose not to support other transparencies in order to manage implementation complexity. It is beneficial to make note therefore of which Transparencies a particular system chooses not to address, because in the case of the Transparency challenge, what a system does not consider is what classifies it within the paradigms. [ML02]

For 2K the Execution Environment abstraction provides directly for Access, Location, and Replication Transparency. Consider that each instance of an Execution Environment is designed to directly support a single user, which means that Concurrency

is less of a concern and thus not directly addressed. Mobility, on the other hand, while addressed directly by use of reflection cannot be guaranteed for all host systems and all user Execution Environments. These transparency design decisions further support the classification of 2K as User Centric. [ML02]

Globus and Legion provide directly for Location, Concurrency and Mobility Transparency, as expected, in order to support parallel processing. It is not clear whether Access can be considered transparent in Grid systems, because the assumption is that most if not all invocations will involve Remote Access, so that Local versus Remote is of no consequence to a Grid. Similarly, Replication Transparency is considered to be of little concern in a Grid System, because the assumption is that most of the available resources will be aligned with a single user request, as opposed to other paradigms that apply a small amount of the total resources to many user requests simultaneously. The absence or minimal treatment of Replication and Access Transparency is characteristic of Processor/Storage Centric systems. [ML02]

The Single Address Space example, Sombrero, demonstrates an implementation that uses unique addresses to provide directly for Access, Location, Concurrency and Mobility Transparency. The unique naming scheme means that indirection is required to provide Replication Transparency. StratOSphere also incorporates support for all five Transparencies considered by the paradigm model. The Transport Layer provides the means to support Location Transparency by supporting several popular remote invocation methods. The Messaging and Repository Layers, along with the Transport Layer, provide capability for Access, Concurrency and Replication Transparency. Finally, the Distributed Object Layer combined with all the lower layers provides Mobility Transparency [WAL98]. Sombrero and StratOSphere are considered Implementation Centric with respect to Transparency based on their support for all designated Transparencies [ML02].

Three additional distributed systems of interest are Project Oxygen, Opus and Globe as described in [Vau02]. These systems are classified as shown in Figure II-12.

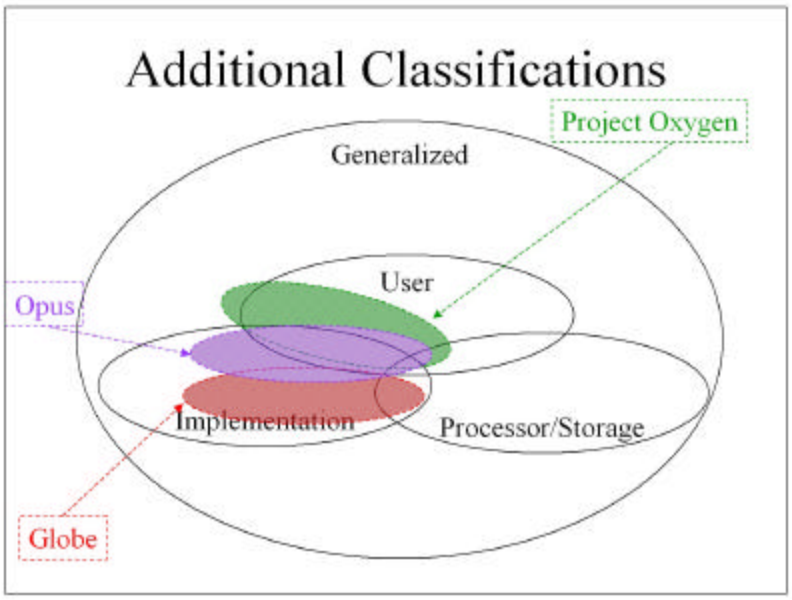

Figure II-12. Additional Example Classifications [from ML02].

Globe uses the object-oriented approach and provides *Globe Objects* that abstract away the distributed implementation detail by providing five functional components or sub objects. Each Globe Object includes a control sub object, a communications sub object, a replication sub object, a security sub object and a semantics sub object. Each sub object provides a high-level interface and performs low-level tasks for the application developer, such as allowing the physical distribution of an object across several machines [Vau02]. Heterogeneity is dealt with indirectly by hiding the details in the sub objects. Scalability is supported directly by the use of the object-oriented approach. Transparency of Access, Location, Replication, Concurrency and Mobility are all provided for directly by component level functionality included in the sub objects. "Each Globe object controls how its state is replicated, migrated, and otherwise spread across machines" [Vau02]. Designed to provide an indirect, middleware approach to building a distributed system, Globe is intended for developers and is classified as primarily Implementation Centric [ML02].

The Opus project extends the UC Berkley WebOS project and is primarily concerned with providing Web services. Heterogeneity is abstracted away by the Opus software overlay and Scalability is provided for but cannot be guaranteed. The

recommended approach appears to be to use a scalable Peer-to-Peer application on top of the Opus middleware in order to assure the overall solution scales. With roots in Web services, a pre-dominantly User Centric method of distribution, Opus is classified as a mix of User and Implementation Centric [Vau02, ML02].

Project Oxygen advances the goal of using a distributed system to provide a computing environment, rather than just a collection of inter-connected computing devices. The environment brings the user's context to the computing environment, making the interaction more natural. Because of this goal, the project is primarily concerned with the mobility of and support for a specific user, and therefore is classified as User Centric [ML02].

Finally, Distributed CAPS (Computer Aided Prototyping System) research at the Naval Postgraduate School defines the requirements for extending the stand-alone CAPS system into the distributed environment. CAPS assists in the development of requirements and timing constraints for real-time systems. The research by [Alm98] suggests a distributed scheduling solution and more interestingly, [Kre99] suggests a distribution architecture. Both research projects aim to extend and strengthen the functionality of the original CAPS tool through distribution. Distributed CAPS is classified as shown in Figure II-13.

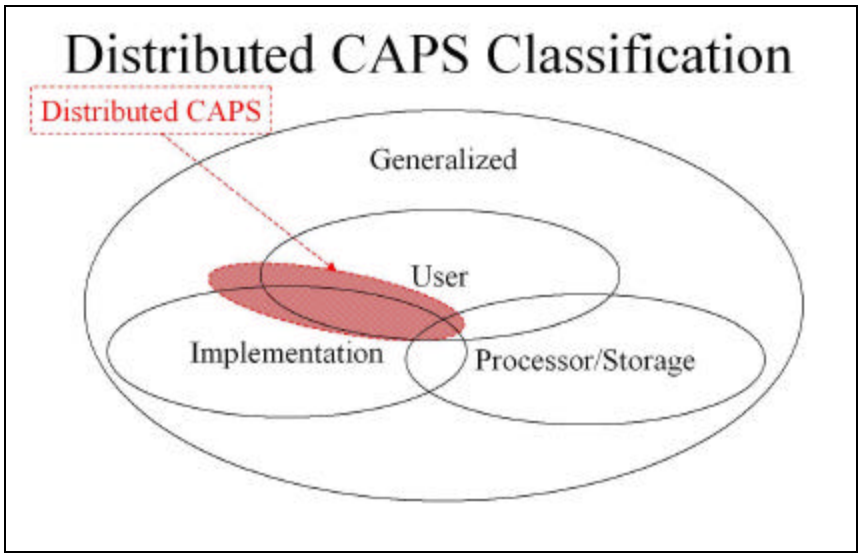

Figure II-13. Classification Of Distributed CAPS.

Heterogeneity in Distributed CAPS is handled indirectly by programming in Java and ADA. Remote invocation is accomplished, despite underlying heterogeneity, by using CORBA. The inherent inability of the client/server architecture to scale well is a concern that is addressed by adding an additional level of servers. Scalability therefore is indirectly supported by sub-dividing the server functionality into two levels, one for communication with clients and one for back-end processing [Kre99]. Dependence on the client-server architecture combines with the use of object-oriented programming to classify Distributed CAPS as User Centric and Implementation Centric. Consider that Opus received a similar classification despite its use of the Peer-to Peer architecture, an inherently Implementation Centric approach.

E. SUMMARY

This Chapter summarized the OOMI concept of an FIOM, highlighting the aspects that lend themselves to distribution. The inherent challenges of the distributed computing environment were described and the basics of software architecture were reviewed. Finally, a model was introduced that provides a mechanism according to which distributed systems can be classified and compared. The next chapter builds on this foundation by suggesting a distributed architecture for collaboration in the construction of an FIOM. One goal of the architect is to achieve a classification as a mix of Implementation Centric and Processor/Storage Centric, according to the paradigm model.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DISTRIBUTED ARCHITECTURE FOR THE OOMI

> "It is a profoundly erroneous truism, repeated by all copy-books and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them. Operations of thought are like cavalry charges in a battle--they are strictly limited in number, they require fresh horses, and must only be made at decisive moments" (Alfred North Whitehead [1861-1947]).

A. MOTIVATION FOR CREATING DISTRIBUTED OOMI

A paradox exists between the requirement to develop complicated interoperability solutions for the decades old systems in DoD's distributed force structure and the existing centralized strategies, centrally managed processes and essentially stand alone software tool sets that are available to manage the creation of the solution. While it is unlikely that we can afford, mentally and financially, a single Herculean effort to mitigate the interoperability problems of the DoD, perhaps we can arrive at the same result by way of a collaborative and systematic process that is affordable.

The Federation Interoperability Object Model defined by the Object-Oriented Method for Interoperability [You02] is capable of representing the inherent incompatibilities in DoD's distributed collection of legacy systems. The OOMI IDE likewise supports mitigation of heterogeneity with automation assistance. A key component yet to be developed is a distributed manor for collaborating on the construction of the FIOM. The compelling reason to distribute the construction of the FIOM is the unavoidable reality that Defense Acquisition occurs in a dynamic and distributed environment. The tools and processes for creating an FIOM must fit this environment and function within the established boundaries in order to be feasibly implemented across DoD.

A non-distributed FIOM generation application would be the least complicated to develop and implement while at the same time would be the most difficult to incorporate as an integral part of a DoD-wide process. The propensity for the scope of an FIOM

construction effort to extend beyond a single Service or Defense Agency is at odds with the lack of scalability inherent in so called stand-alone applications. The horizontal integration of components required for FIOM construction introduces concurrency complexity that adds to the difficulty of accomplishing the task. The stand-alone solution therefore requires a separate collaboration solution to facilitate the process of constructing an FIOM in the DoD environment and is not the most promising approach.

Providing a collaborative environment surrounding a central database and associated client application is also less capability than the scope of DoD problem requires. The database approach solves the concurrency and integration concerns by centrally locating the FIOM while under construction, but such a solution would likely be of proprietary origin and use the client/server architecture, thus introducing new concerns regarding scalability and openness. The interoperability made possible by the FIOM must persist over time and so must the FIOM generation tools and FIOM component storage solutions. For this reason a database application is considered too restrictive for long-term storage and management of the FIOM.

Our distributed solution allows for maximal flexibility of participation without introducing additional infrastructure and support requirements. By using existing Internet connectivity and readily available computer systems to provide a collaborative environment for the generation of the FIOM, our solution persists in time without procuring and maintaining a monolithic system specifically created for the OOMI. A distributed environment for the FIOM facilitates construction on a first-come, first-serve basis without necessitating establishment of standards ahead of time. The chronological population of an FIOM defines the standard incrementally and only as completely as is required by the current scope of the Federation.

Designing the distributed environment for automated creation of an FIOM is an exercise in creating a distributed system and so all the challenges of distributed systems apply to the design and implementation of the Distributed OOMI. This architecture for the Distributed OOMI is the result of a methodical exercise in matching requirements to capabilities in order to provide a solution while avoiding dependencies that limit the potential of the solution to function in its intended environment and persist in time. We

focus on providing capabilities that support the processes of the OOMI, rather than creating a distributed version of an inflexible data set based on literal definitions. The purpose of the Distributed OOMI is to limit the amount of future effort required to build and use a distributed FIOM. Distributed OOMI is therefore an enabling technology that facilitates the creation of a more powerful enabling technology, an FIOM. From the FIOM, the generation of interoperability solutions for stove-piped DoD systems is possible, which introduces a rapid and sufficiently simple methodology for deriving significant operational capability from combinations of previously independent and incompatible resources. Producing the means and process to enable new operational capability through leveraging existing resources is currently possible only after considerable expenditure of thought and engineering effort. With the OOMI and a distributed system to support the creation of an FIOM, we provide a collaborative environment for the creation of a resource that may lead to any number of innovative combinations of legacy systems.

B. THE LAYERED ARCHITECTURE

The architecture as shown in Figure III-1, is layered to support multiple abstractions of complicated distributed systems concepts. From top to bottom, the layers are termed *Application*, *Storage* and *Peer Networking*. In the application layer we find the GUI-based IDE for FIOM construction, which runs in user space on the participating Distributed OOMI host systems. Below the applications layer is the storage mechanism, which may or may not be directly incorporated into the IDE, but is at least available to the application layer via an interface. This is the most important layer in terms of supporting the distribution of an FIOM, and so it includes two types of storage termed *Local* and *Remote*, which are handled separately and with different solutions. The storage mechanism sits on top of a peer-to-peer networking layer that provides access to the distributed peer participants. The existing Internet is available below and via the peer-to-peer layer.

The desired logical result is for the layers to allow multiple instances of the Distributed OOMI to present a single perception of the FIOM under consideration, despite its lack of physical existence in a single location. The architecture is primarily

layered, but within the layers the architecture makes use of object-oriented techniques, as well as flat file storage. We add the peer-to-peer networking in the communication layer on top of the existing physical packet switching infrastructure in order to gain the benefits of multicasting and abstract some of the complexities of network collaboration. Because multicasting is not generally available on the existing Internet infrastructure, a peer-to-peer overlay network that implements multicasting is preferred over other approaches.

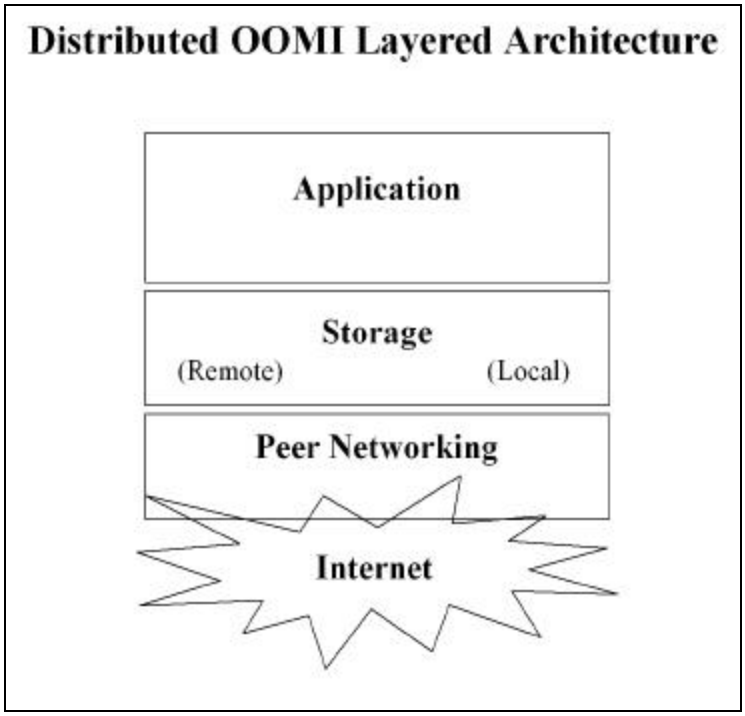

Figure III-1. Distributed OOMI Architecture

This research provides in Chapter IV a description of some of the implementation concerns for the layered architecture presented in Figure III-1. The main focus of this research, however, is the Distributed OOMI architecture itself and in particular the details of the Storage layer of the architecture. The Storage layer is the glue that leverages the Peer Network layer and power of application programming to make distributed collaboration for the construction of an FIOM an easier task. At some point in the future, the peer networking capabilities desired for the architecture may be provided by the host operating system and the network infrastructure. Whether part of the host operating system or approximated by middleware, the concepts of the architecture are the same.

The remainder of this chapter describes the functional details for each level and further develops the pictorial representation of the Distributed OOMI layered architecture. Section C covers the enabling technologies and how they are applied to the Distributed OOMI. Section D introduces the details of the data structures used in the Remote portion of the Storage Layer. Section E continues with the details of the Local portion of the Storage Layer. The Distributed OOMI approach to mitigating the challenges and transparencies of distributed systems are discussed in Section F. We conclude this chapter by discussing the aggregation of all the parts of the architecture and assign a classification to our approach so that it may be compared meaningfully with other distributed systems research efforts.

C. CONCEPTS THAT SUPPORT DISTRIBUTED OOMI

Distributed OOMI is made up of a collection of existing techniques, applied in a slightly non-traditional manor to yield the desired distributed collaboration result. To facilitate the use of existing technology, an abstraction of the OOMI FIOM (introduced Section II.A.2) must be formed that relates the FIOM in terms of the technology to be applied. The FIOM essentially provides storage for a few basic types of information. We partition these types of information into two categories termed *light-weight* and *heavy-weight* components.

The light-weight components provide storage for the complex association, specialization, generalization and logical containment relationships that are required to describe the structure of an FIOM. Storage for these complex relationships is accomplished by way of simple references that are stored as data within the lightweight components. These references point to other light-weight and heavy-weight components such that logical containment and hierarchical inheritance relationships for the FIOM are persistently stored in a distributed manor.

The heavy-weight components are required to represent the Component Systems, the Federation Representation of Component Systems and the Translations that permit the association of a Component Class Representation (CCR) with a Federation Class Representation (FCR). These components are used during the Translator Generation phase of the OOMI to produce the wrapper-based translators that enable interoperability.

A complete representation of the FIOM can be formed by aggregating heavy-weight components and Translations discovered while following chains of references between the storage components.

Many of the components of the FIOM can be represented by simple data structures that are named and contain only other data structures. Recall from Chapter II that a Federation Entity (FE) contains one or more Federations Entity Views (FEV) and that FEVs in turn contain Federation Class Representations (FCR)s and so on. Physical containment is not as important as the logical containment relationships between FIOM components, because the payloads of most FIOM components consist primarily of other FIOM components. The functional requirement of representing these containment relationships is satisfied if we create a data structure to capture and store logical relationships thereby preserving the means to re-construct physical containment relationships. To support distribution we fashion the data structure so that each FIOM container is represented as a physically unique node. Containment relationships are represented by links between the nodes and the overall result is an abstraction of the FIOM that is a tree, as shown in Figure III-2. There are several types of tree nodes, each of which approximate a corresponding FIOM component and maintain the containment relationships by storing parent and child references to related nodes. When persistently storing the tree model, each node is provided an independent data structure and unique storage location. By storing nodes individually, the containment relationships are captured in a verbose, but diffuse collection of relatively compact structures, and so we call these structures the *light-weight* distributed components of an FIOM.

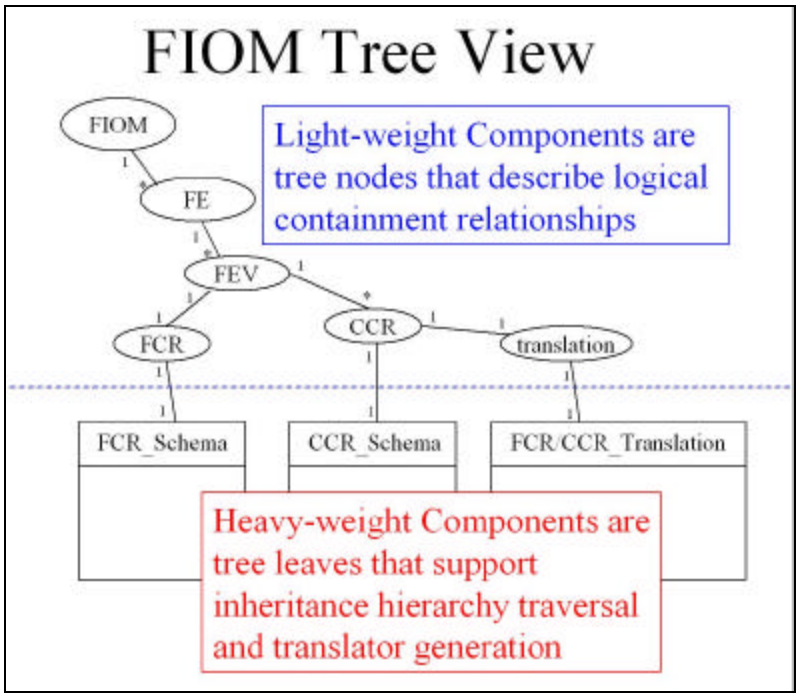

Figure III-2. Logical Containment Relationships Between FIOM Components.

The other functional requirement for the OOMI FIOM is the storage of what we term *heavy-weight* components. The heavy-weight components satisfy the need for the FIOM to store potentially large objects associated with the interoperability translations and are the leaf nodes of our tree abstraction. For example, FCRs and CCRs both contain heavy-weight components that represent Schema, Syntax and Semantics objects. The physical makeup of these and other heavy-weight components could be serialized Java objects or relatively large text files. When compared to the light-weight components, the heavy-weight components are much larger in size, complexity and capability. These more capable components are collected from distributed storage, when needed, to construct an FIOM Lattice, such as the example in Figure III-3. The references provided by the light-weight components facilitates searching for paths between CCRs and FCRs, but the real interoperability work is done by the heavy-weight components that combine to form an FIOM Lattice.

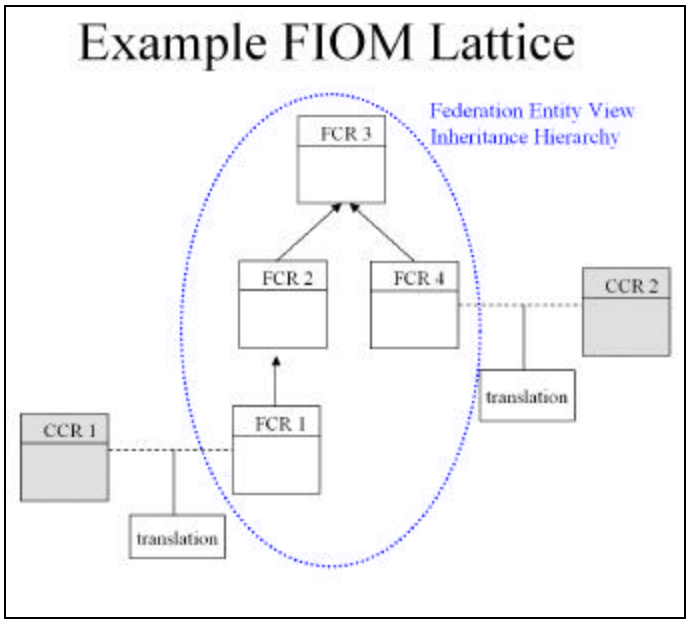

Figure III-3. FIOM As A Lattice.

A lattice represents a traversal of the FIOM inheritance and translation relationships, and can be used both for registration of new Component Systems and during run-time by OOMI Translators (Section II.A.3) to facilitate interoperability between specific Component Systems. Figure III-3 shows an example of an FIOM Lattice that describes a path between systems registered via CCR1 and CCR2. An OOMI Translator for these two systems would use classes and translations discovered while traversing the path between CCR1 and CCR2. A traversal between FCRs along the path is a special case of translation that may be as simple as casting from one to another using the implementation language's native typecasting capability. This portion of the traversal may be more complex involving up-cast and down-cast transforms that are similar to the FCR-to-CCR translations required at the edges of the Federation.

Section E discusses specific implications of an FIOM Lattice in further detail. For now, the key consideration is that heavy-weight components provide complex capabilities, are relatively large and infrequently used relative to the light-weight components. Our abstraction of the FIOM using light-weight and heavy-weight components permits the use of two existing and rapidly maturing technologies, XML and peer-to-peer networking, to simplify the architectural requirements of storage for the

Distributed OOMI. Before we can introduce the data structure for Distributed OOMI, though, dependencies on capabilities facilitated by XML and peer networking must be established and described.

1. XML and Object Mapping

The intent of Distributed OOMI is that an FIOM be represented by a shared model, accessible by any number of application environments. Thus the organizations collaborating to construct an FIOM are not necessarily tied to a single tool or computing environment. Using the Web as an example, this idea is similar to the way web authoring tools and web browsers both work with HTML formatted text. Browsers and authoring tools are applications written for many operating systems and in several programming languages, but can openly access HTML. There are several ways to view and implement Object Mapping; we focus on the usefulness of data binding for Distributed OOMI.

There are two developments that assist us in implementing Distributed OOMI and these are the Extensible Markup Language (XML) and Data Binding. XML is useful as a portable data serialization method that provides text-based storage with embedded markup information describing the format and purpose of the data. Data Binding defines the relationship between data serialized for storage as an XML document and memory resident programming language objects. Choosing XML as the primary data storage format provides the benefit of mitigating several aspects of the heterogeneity, scalability and openness challenges. [AAC+01]

The light-weight data storage occurs in XML format, an open and portable W3C text based data formatting standard. ASCII text is portable to the vast majority of automated systems, though the usefulness of text only storage is sometimes limited. Data stored and formatted with XML is more useful than plain text because of the embedded XML meta-tags and the existence of high-level programming language Application Programmer Interfaces (APIs). The extensible part of XML is the ability to define what the meta-tags are and what they are used for.

Data Binding is the process of converting between object-oriented memory resident data structures and persistent open format serialized storage. XML as an open

format is valuable because of its ability to capture formatting for the data within the serialized storage. We desire to avoid the use of object-oriented serialization methods for persistent storage because the XML serialization improves openness in the collaborative environment.

Because Data Binding is a serialization technique, the object-oriented version of the data can be implemented in any object-oriented language, but we choose Java. The open XML serialization format improves support for heterogeneity by not restricting the object-oriented portion of the implementation of the OOMI tools to a proprietary or otherwise closed data format. Rather, future OOMI tools need only implement Data Binding from the XML storage format to any memory-resident format of choice. The open source collaboration surrounding XML provides many of the API components needed to implement extraction of data from the serialized format. Several such APIs exists for the Java language and are recommended for use by Distributed OOMI, as discussed in Chapter IV.

2. Peer to Peer Collaboration

We desire the benefits of a single shared bus network in order to limit the complexity of the network interactions between systems collaborating in the construction of the FIOM. At the same time, the distributed solution requires dependence on the existing Intranet. We therefore arrive at the desired functionality by imposing abstractions on top of existing network infrastructure in such a way as to enable the techniques that we desire for our persistent storage and distributed collaboration. We assume that the complicated network infrastructure of the current Internet environment is not sufficiently capable of providing the desired functionality to support Distributed OOMI without the addition of our proposed layers of abstraction.

For transparent collaboration, we use a peer network to establish the specific context for the collaboration and thereby decouple location from identification of the participating host systems. Currently context is determined in the Internet completely at the user level. There is no specific support for grouping all network activities that can be considered to exist because of a specific user level context. For example, consider that a user may generate network traffic in support of research and collaboration surrounding

the purchase of a house. The user would likely conduct many Web searches, send email to certain realtors and homeowners and perhaps apply for a mortgage via a Bank's Web interface. In this example the user aggregates the results and provides the context. This model works for single and even small groups of users, but does not scale to support large numbers of users and large numbers of communications and formats required for effective distributed collaboration. The peer network concept allows the inclusion and exclusion of hosts from the network based on the context of the communication between the hosts. This is the exact capability we desire for distributed collaboration.

Consider the following example that illustrates the usefulness of Peer networking in devising a layered abstraction that supports our end goals. As shown in Figure III-4, the underlying networking infrastructure is complex. The Internet is made up of conglomerations of Local Area Networks, joined by routers. Each LAN may be several router hops away from every other LAN and host systems are abstracted from the rest of the Internet by LANs.

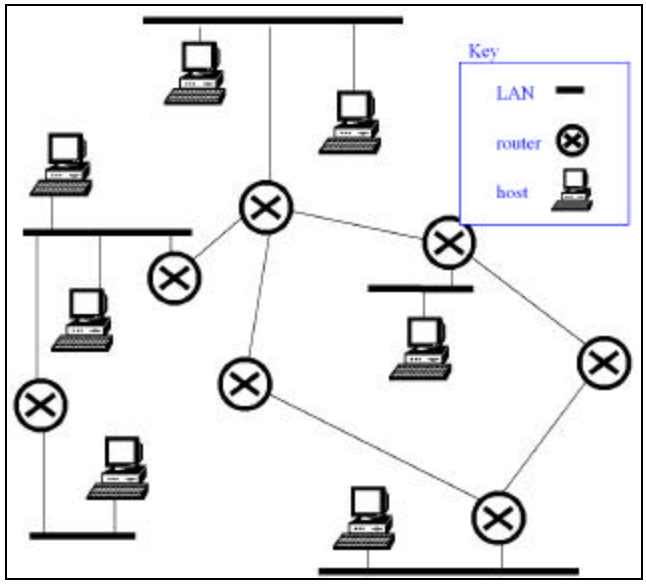

Figure III-4. Representation Of Normal Network Complexity.

Routers develop and maintain a consensus with respect to which LANs can be reached via any given router or essentially, the path to a specific LAN. The LAN is the final routing destination for a packet switching network, and the LAN is responsible for

dissemination of messages to the host system. The system of interconnected routers develops path information to hosts, or more simply, routers determine *where* hosts are located in the network. Hosts are uniquely identified by their IP numbers, which are the names used by the Internet to determine *whom* any given message is addressed to or from. In the Internet Protocol, the *who* and *where* are strongly coupled.

The most difficult aspect of distributed computing is overcoming the dependence relationship between identity and location. Systems must be uniquely identified as well as have a relative position in the network to allow routing and delivery of messages and data. When both identity and relative location of a distributed application's participants are dynamic with respect to time, using only IP to keep track of the *who* and *where* is more difficult than need be. The peer networking environment relieves much of the difficulty by further abstracting the *where* portion from the application. By introducing a Peer overlay, we reduce the complexity to something similar to what is shown in Figure III-5. Notice that the peer overlay in our example is sparsely populated with links. This is a characteristic we generate during formation by way of imposing coupling requirements as part of the algorithm that forms the peer network overlay.

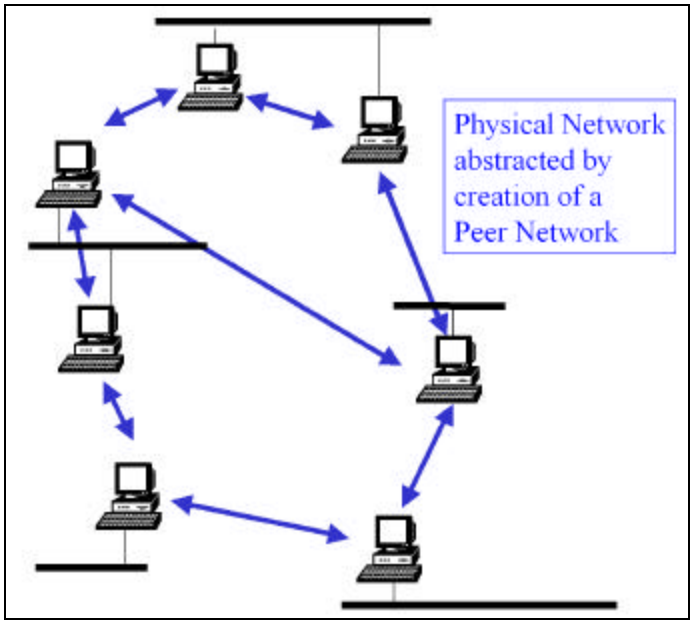

Figure III-5. Peer Network Complexity Example.

The location of the other hosts is less important in the Peer to peer network. Hosts are still aware that access occurs via a LAN and the name or identification for the target host is necessary. The peer network abstracts some of the complexity at the application level, but the coupling between who and where is still strong. With just a peer network overlay the primary benefit is the ability to multicast messages, a capability not yet pervasive in the Internet. Because of multicast we can say that the peer network is less complex than the original network with respect to determining location of a host. Determining who with respect to a network communication is still of paramount importance, however, and the identity of a host remains coupled to its location in the network, despite the reduction in coupling gained from multicasting.

The implementation of the peer network is key to the success of the Distributed OOMI, but we need one more restriction on the peer-to-peer model to reach the desired degree of abstraction at the application level. In Figure III-6 the peer network participants remain attached to the physical infrastructure, shown in Figure III-4, in the same manor as other non-peers. Additionally, the peer traffic traverses the complex infrastructure in the same fashion as non-peer traffic. Consider that we can logically arrange the peers using multicast so that network transmission is abstracted in such a way as to create a *logical bus*. A logical bus is modeled after a physical layer bus, such as an Ethernet Bus, and allows the conceptual existence of a logical *collision domain* with respect to the abstract bus. The term collision domain means the same as it does in the physical level of networking in that the hosts compete for the transmission media and may transmit simultaneously causing a collision. Our collision domain abstraction is implemented at a higher level of the networking protocol stack and will not be subject to the collisions of a shared media, but will maintain the ability to disseminate traffic simultaneously to every system connected to the shared bus. If we consider the peer networking environment to be organized similarly so that the peers are basically sharing a single logical bus, then we can assume that all the peers will eventually hear all the network traffic. Requests for distributed storage or retrieval are simplified greatly because the *who* and the *where* are decoupled in a shared bus architecture.

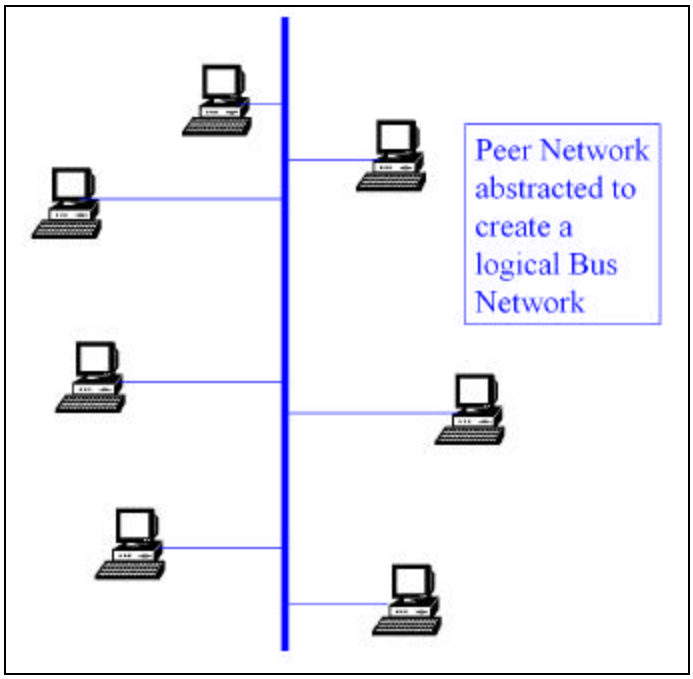

Figure III-6. Logical Bus Network Abstraction.

The two key results that come from creating a shared bus with the peer network overlay are the removal of location and identification coupling from application level inter-host communications and the ability to send to every node by simply placing traffic on the logical bus. To demonstrate the value of the logical bus concept consider the following description of our intended use of a logical bus created for Distributed OOMI.

The physical propagation of data packets remains as it is shown in Figure III-7. The detail of the Internet is abstracted by the use of a cloud symbol. A router connects each LAN to the cloud, and we assume that within the cloud there exists at least one path between the cloud edge routers over which data packets can flow, thereby connecting the two hosts, via the Internet. The idea of the cloud is an important abstraction that we wish to make use of in Distributed OOMI. For clarity, we consider a cloud to be any complex entity with irregular and difficult to define edges that is, for intensive purposes, functionally opaque. The Internet cloud is a concept that is based on the fact that paths between source and destination are dynamic. The routers of the Internet will determine the path, thereby making the actual paths of any particular data flow essentially invisible from the perspective of the sender and the receiver.

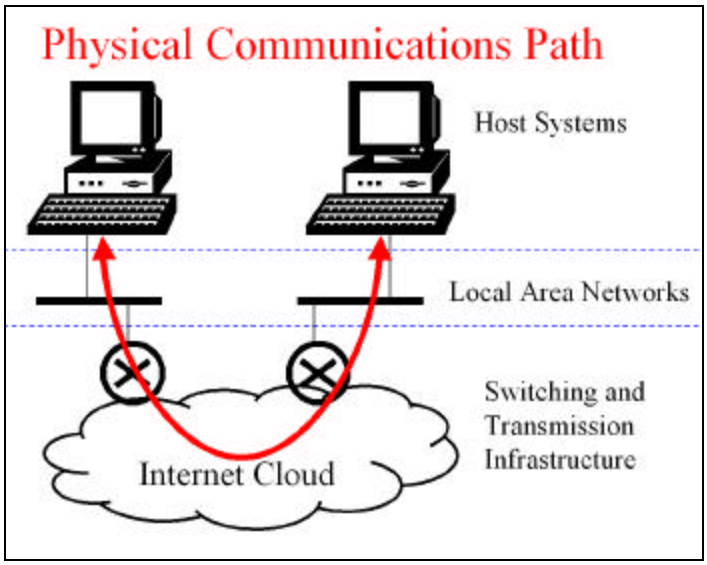

Figure III-7. Physical Propagation Of Peer To Peer Network Traffic.

The normal concept with Peer to Peer involves the communication shown in Figure III-8. The peers are aware that the network is being used to communicate with other peers, and in general the other peers are known and referenced specifically. With a peer-to-peer network overlay in place, host systems are aware of network communications, but the details of routing and domain name or IP resolution are contained within the peer networking layer. The peers are uniquely identified in the peer network and this identity is used to direct messages to the distant peer without specifically addressing the existence of the underlying network infrastructure. From both the sender and receiver perspective, the message being transmitted may have never left the Local Area Network, though we know the packets traversed some unknown number of hops on the physical Internet. When the abstraction is expanded to our concept of a logical bus, the network transmissions to distant peers exhibit similar reduced complexity. Sender and receiver are only aware that communication occurred via the networking level of the protocol stack and have no interaction with the underlying dependence on IP and the Internet.

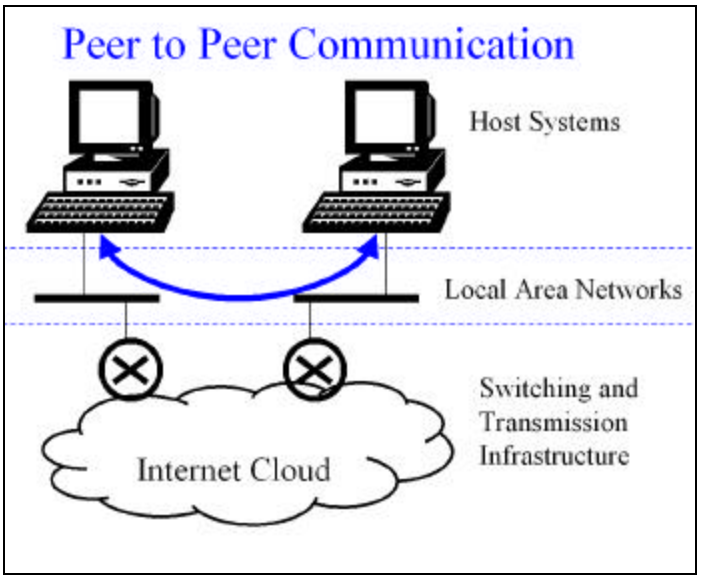

Figure III-8. Logical Communication Path.

This approach limits the complexity at the application layer by completely abstracting the lower-level networking functions below the peer network overlay. More importantly, the addition of a peer layer reduces the number of target hosts to only those participating in the peer network, thereby creating a sense of contextual participation.

In our view of the peer network, a single participant in the peer network is aware that the network is being used to access the other peers, but the identity and location of the other peers is not important and not available. Requests are made to a *Peer-Cloud*, and replies come back from the cloud with the source and even the destination being of little consequence within the context. The reason being that the context of the communication is established by the formation of the Peer-Cloud. Peers join a specific Peer-Cloud with the expressed intent and purpose of participating in the context for which the Cloud was created. In our case, participation in creation or use of an FIOM is the reason and context for the Peer-Cloud.

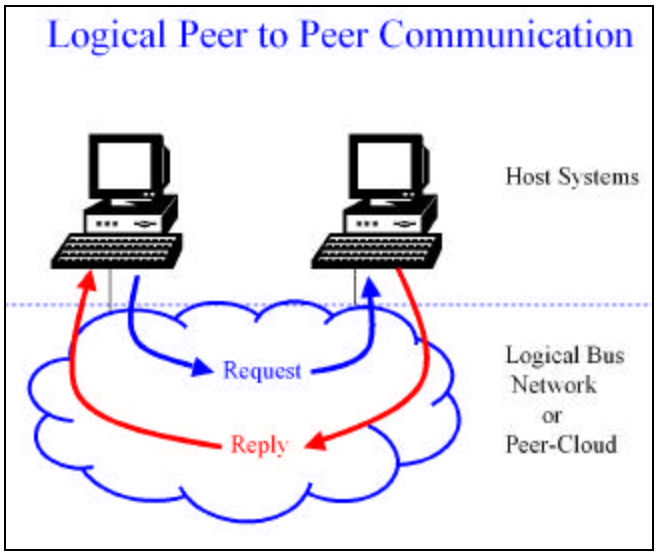

Figure III-9. Peer-Cloud Example.

Conceptually we are able to request in a generic fashion, within the context of the Peer collaboration, and yet receive a specific reply. Recall that every host node in the peer collaboration will conceptually receive the traffic sent by every other node, and so every host may potentially generate a reply. This logical bus behavior is accomplished through multicasting and relaying over a sparsely connected peer network, rather than broadcasting on a strongly connected peer fabric. The implementation details of the multicasting will provide the means for the divergent propagation of requests and the convergent propagation of replies. The method is intended to be more sophisticated than a simple broadcast and should make use of a breadth-first approach to propagation on the sparsely connected peer network, such as the network shown in Figure III-5. The closest neighbor peers receive the original request and then make a request on behalf of the originator to a neighbor peer not yet visited in the peer network concerning this request. In this recursive approach, the number of Requests increases until all nodes in the peer network have been visited, and the number of Replies decreases as each level of recursion completes. The originator therefore receives a number of Replies not more than the number of Requests. If the first request is made to a single neighbor, there will be only one reply received. One very important difference between a physical bus and our logical bus is the time it takes for messages to propagate. In the logical bus there is

no expectation of universal time, and so relative time must be used to determine delay in response to a request. These issues will be discussed in greater detail in Chapter IV.

The relay of requests and replies is the final portion of our abstraction and alters our perception of the Peer Network Layer for Distributed OOMI so that requests and replies to the logical bus can occur in both directions. A peer connected to the bus may send and receive both requests and replies as shown in Figure III-10.

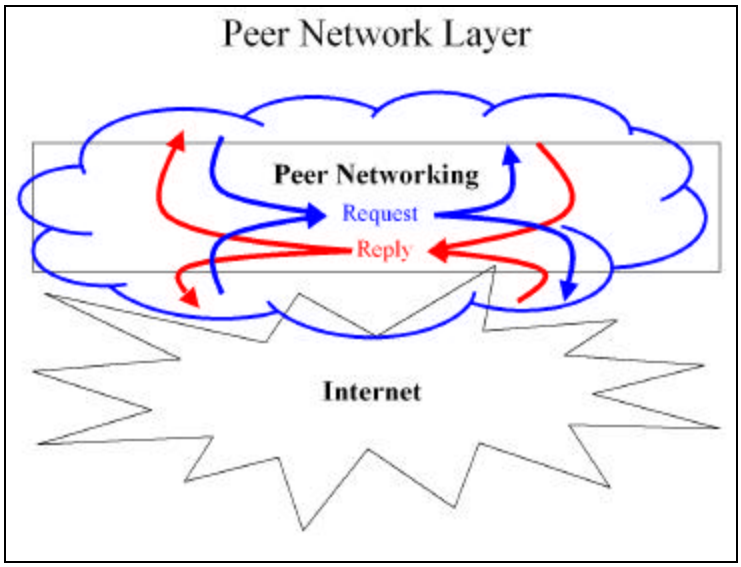

Figure III-10. Peer Cloud For Distributed OOMI.

There are many complex issues associated with providing the logical bus peer network. The lower level middleware providing the peer-to-peer service handles many of these issues including reliable transport, maintenance of the peer topology and the interface with the underlying physical topology. Research in peer network overlays is ongoing at many institutions around the country and we anticipate that a peer-to-peer solution meeting our requirements is possible to implement for the Peer Networking layer of Distributed OOMI. Though existing peer network overlays may not provide entirely for the logical bus functionality, the behavior may be approximated for the purpose of demonstration. In fact, the functionality desired could be minimally emulated by a client/server or other distribution architecture; however, peer-to-peer is the most appropriate and beneficial, in terms of distribution capability, of the architectures currently available. The main reason for choosing peer-to-peer is that it scales better than

other architectures and is less complex to implement, as it requires no additional external server support and no additional hardware. The creation of the peer overlay software that provides for the complete logical bus functionality is not simple; however, a software solution using an overlay network is much less effort and expense than an entirely new operating system, is more powerful than a communications protocol, and examples of peer-to-peer overlay solutions already exist and can be extended or patterned after to create the desired functionality. Regardless, the issue of implementing the logical bus peer network must be left for future work, despite our assumption that such a capability is required for Distributed OOMI.

D. A DISTRIBUTED DATA STRUCTURE FOR OOMI

As mentioned previously, XML is the data serialization method of choice and a logical bus peer network is the method for accessing and transferring components between hosts participating in the Distributed OOMI collaboration. The next subject is the description of the data structures for distributed storage of light-weight and heavy-weight FIOM components.

Every method of distributed storage must provide a data format, the means to write data, locate previously written data and finally read data. Of these concerns, the task of locating specific data is the most difficult in the distributed environment. Reading and writing are less difficult, because these functions often deal with a specific location and format and so the scale of the distributed data storage mechanism does not directly affect their behavior. The breadth of the storage space is abstracted from the reading and writing activities, which are normally operations of a particular host more so than a generically defined distributed system function. Data formatting is also often closely associated with system requirements or decided indirectly based on limitations inherent in the chosen implementation language or platform. The decisions made concerning these four basic data storage functional requirements will impact the overall effectiveness of the distributed system. The data structures described in this section are designed to produce a scalable and open data storage solution for the Distributed OOMI. Before introducing the storage methodology a few key concepts must be introduced.

1. Self-Similarity

A scalable distributed system exhibits the characteristics of self-similarity and load distribution according to a Power-Law [FF02]. Self-similarity for an effective distributed system means the composition, structure and behavior of the system, when observed at some level of logical abstraction, are similar to observations at a higher or lower level of abstraction [FF02]. The load placed on such a system will be distributed according to a Power Law relating relative load to number of occurrences, such that heavy loads will be relatively few in number, and lighter loads will be orders of magnitude greater in number. The Internet's packet switching infrastructure is a distributed system that demonstrates effective scalability and that exhibits both characteristics [FF02]. The composition, structure and behavior of the Internet are similar when considered at the Autonomous System level, regional ISP, local ISP, etc., all the way down to some atomic level, below which no further abstractions exist [FF02]. The distributed loading on the Internet is produced by a continuum with very heavy demand on behalf of a few nodes at one extreme and very light demand by the vast majority of nodes at the other [FF02]. Based on the findings concerning the Internet and the fact that the Internet is a distributed system that scaled well, we suggest that any distributed data structure designed in this manor will in turn scale well. To facilitate scalability of our architecture, the data structure for Distributed OOMI attempts to approximate the self-similarity and Power-Law load distribution exhibited by the Internet.

The OOMI describes the functional requirements for creating and storing the heavy-weight components of the FIOM as Java objects, but the storage format for the light-weight components, as well as the location of the storage and methods for reading and writing are all open to interpretation. Therefore, we can create a distributed data structure that functionally supports persistent FIOM storage by devising specific methods for formatting, locating, reading and writing FIOM components, without violating the requirements of the OOMI. If we use the two types of FIOM components, described at the beginning of this chapter, to distribute the storage of the FIOM by generating self-similar components such that the majority are light-weight components and the minority

are heavy-weight components then we facilitate an approximation of the desired Power-Law relationship and self-similarity properties in the terms of our data structure. Additionally, this approach to the data structure provides a high degree of selective cohesion with minimal coupling and complexity.

The format for the light-weight components is designed to produce storage components of FIOM data that capture and exhibit the self-similarity characteristic of the logical Federation containment relationships. The data storage components contain information about a particular piece of the FIOM and references or links to the other components to which a storage component is related. Recall from Figure III-2 that light-weight components are comparable to tree nodes and are related to one another by parent/child references. For example, a Federation Entity (FE) data storage component would be named for the particular FE in question and contain references to sub-components as well as the FIOM to which the FE belongs. Figure III-11 shows a generic representation of the structure of light-weight components. The actual structure of each specific type of light-weight component may be unique because we wish to preserve flexibility with respect to the data storage requirements for the type of component in question.

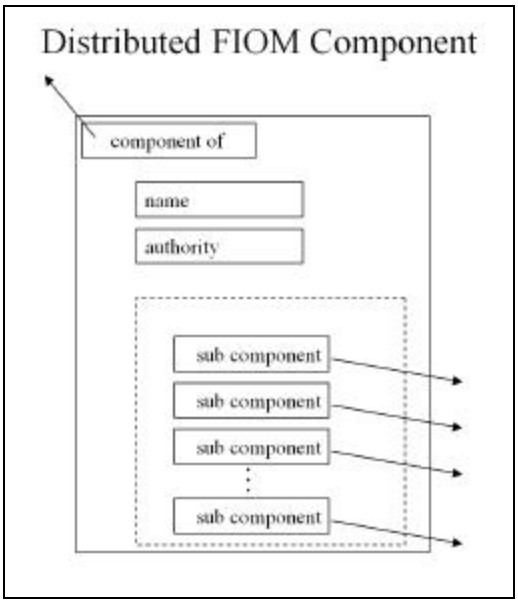

Figure III-11. General Model For A Light-Weight Component.

These minor differences prohibit the use of a single light-weight component for every FIOM equivalent, but the entire set of components considered together exhibit self-similarity regardless.

2. Openness

The light-weight components are ultimately stored persistently as text in a flat file format. ASCII text and the concept of a file are the most pervasive of existing storage formats and both have persisted throughout the short history of computing. Choosing a text based file format is paramount because it allows almost any architecture to be used; for instance, consider that the Web uses text based files and the client-server architecture. Additionally, the concept of a file is the primary logical persistent storage construct for the host systems that will support Distributed OOMI and the format for persistent storage of the heavy-weight components.

The Distributed OOMI layered architecture makes use of flat text files for storage of components; however, this technique for storage is abstracted from the application and the peer network. Despite the continued reliance on files by existing operating systems, the usefulness of the file paradigm in a collaborative environment is extremely limited. Because of this, we use files for storage but do not allow the files to be directly manipulated by the users or applications within the context of an OOMI peer network. At the same time, however, the files could be available for use in other contexts, such as Web Browsing, because the light-weight components are based on an open and pervasive text format and are stored on the physical storage media associated with a participating host system in a simple directory structure. The heavy-weight components, which are serialized Java objects, are less useful outside the context of the FIOM, but are also stored in a similar directory structure and are also accessible outside the context of the Distributed OOMI application.

3. Location and Naming

The location of distributed data is difficult to manage and normally leads to concern over replication, concurrency and versioning of stored data. There exists a surprisingly simple way to reduce the complexity of storing data in a distributed manor. Consider that there are only two places relative to any given host system that the data can

be located. The data is either stored remotely or locally and the challenge for a distributed system is to handle both cases, as well as distinguish between the two. We start by assuming that all data held in persistent storage is considered to be exclusively Remote. Data for consumption or calculation is considered to be exclusively Local. Data is retrieved from remote storage and assembled locally to support specific consumption requirements.

To illustrate what is meant by these assumptions, consider the World Wide Web as an example of this storage model. The Web uses text based distributed storage and the client/server architecture to access the storage components. A browser application assumes that the data required to render any particular web page must be collected from a remote source in order to display a page locally. Another distributed systems that stores text data is the Domain Naming Service (DNS). The important concept borrowed from DNS is the idea of authority. One DNS server is the authority for a particular Domain Name resolution, though other entities may be able to resolve the name as well. There always exists the possibility of checking with the authority if other sources fail to yield an answer. For Distributed OOMI, the organization that creates and introduces an FIOM component shall be the authority for that component, though over time the component will be available for consumption from other sources within the peer network.

Often considered part of the location information for a particular storage component of data is the name by which the storage component is known within the system. The OOMI includes a Federation Ontology that provides unique naming for all Federation Components. Because we store each Federation component as an individual flat file, the file storage is named according to the Federation Ontology name, thus preventing naming collisions. Further, we have decoupled the name from the storage location, because all requests go to the peer cloud which is non-location specific. The result is the ability to request by unique name any component of an FIOM via the peer-cloud. The ability to request a component by name reduces the complexity of the reference mechanism required as part of the light-weight component structure. To reference parent or children components, a light-weight component need only store the name of the target component. Heavy-weight components are also named and requested

in the same manor, so that light-weight components can contain references to associated heavy-weight components by storing just the name of the heavy-weight component.

4. Remote Storage and Local Storage

Within the Storage layer we define two specialized data structures: one for Remote Storage, and one for Local Storage. Remote storage as we discussed above is for persistence of storage and its data structure addresses related concerns. Local storage is used to support the Distributed OOMI application layer and ultimately the application user, and so its structure is centered on access and usability. The Remote form of storage provides distributed persistent storage by way of managing a collection of files, stored throughout the network on the disk drives of participating systems as XML files. The Local form of storage makes use of XML Data Binding to build an FIOM lattice within the application host's memory space, primarily consisting of Java objects.

The Remote storage data structure is the collection of components that make up the distributed storage capability. This type of storage derives from the ability to represent the logical container relationships and inheritance hierarchy of an FIOM as individual components. The Local data structure is the aggregation of select components that make up the local or working memory storage capability. Local storage is necessary because we desire to traverse the FIOM in its assembled lattice form in order to develop Translators for specific Component Systems. Figure III-12 shows the two types of Storage layer structures. The Local storage data structure is one result of work by Lee [SCL01].

The fundamental difference between local and remote storage is the data structure model. In actuality, location has very little to do with our terminology because all of the remote components of interest may be physically stored on the local host machine. These components whether collected from across the peer network or found on the local hard drive must be combined into the Local data structure to be used efficiently by the Distributed OOMI application.

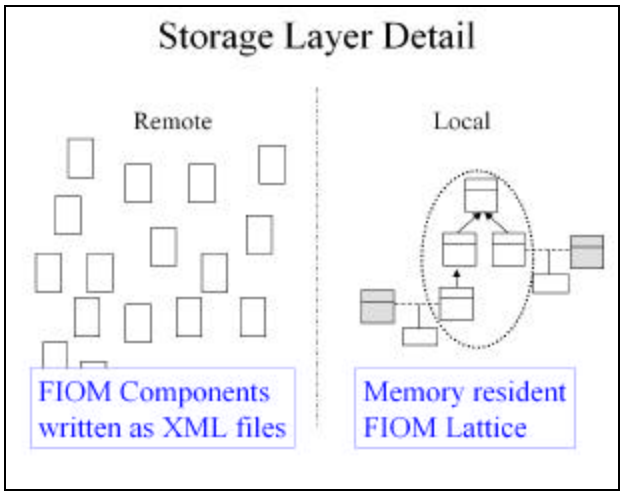

Figure III-12. Detailed View Of The Storage Layer.

The remote case provides for loose coupling of data while in its persistent form. The local case provides a highly cohesive FIOM lattice that caters to the needs specific to user interactions. Moving between the two cases is accomplished via a simple policy: *write once/read only thereafter*. The logical containment relationships are stored as text within light-weight components. The light-weight components are dispersed throughout a network of participants in remote storage. Working with such loosely coupled and highly distributed data is difficult however, and so we fuse the distributed light-weight components into a local representation to simplify the application level interaction with the data. The full view of an FIOM is created when the light-weight components are collected and aggregated into a Local storage FIOM data structure. The heavy-weight components (Java classes) are necessary for the inheritance relationships and are stored separately from the light-weight components, as they need not be migrated as often as the light-weight components. We know that a heavy-weight component is needed because a traversal of an FIOM lattice identified an inheritance relationship or translation requirement. When a heavy-weight component is needed, the portion of the light-weight model that stores the references to the heavy-weight component supplies the name of the target heavy-weight component. As with the light-weight components, the name is the only information required to retrieve the heavy-weight component from its remote storage location. Recall, there are no physical dependencies between the storage components and so they truly are loosely coupled in Remote storage. Figure III-13

represents the self similar storage components of distributed storage as a field of disjoint components.

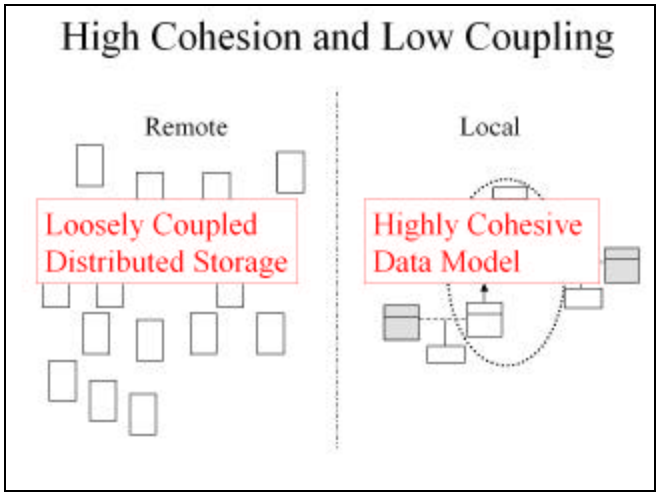

Figure III-13. Coupling And Cohesion.

The loosely coupled Remote storage is exactly what we desire for distributed storage. No component depends on another and if several copies of a component exist, one is considered just as good as any other. Concurrency is not a concern because once a component is written to Remote storage, it is designated read-only thereafter. Failure handling is addressed by caching multiple identical copies at peers around the network and by allowing any Peer-Cloud participant to respond to any peer network request with a cached copy of the requested component.

The Remote storage components are therefore used to append information to a collaborative context, which in our case is an FIOM. Rather than replacing or editing data held in a single file or location, we edit and replace within the context of an FIOM by adding more components or new components in the form of new Remote storage files. When a stored component must change, the old component continues to persist in remote storage and the new component is introduced, starting what in Local storage is seen as a new branch in the FIOM lattice. By simply appending a component that contains new or different information, we complete, or make corrections to, the model over time, without concern for finding and replacing or removing 'old' components that are spread throughout the distributed storage. During the combination process that occurs when

forming a Local representation of an FIOM lattice, the new Remote component is assimilated into the local view of the FIOM and appears as a new branch in the lattice. Components to be written to Remote storage are created by the application and so, as shown by the arrow in Figure III-14, data from Local storage that is to be stored persistently is moved to Remote storage by way of an application process.

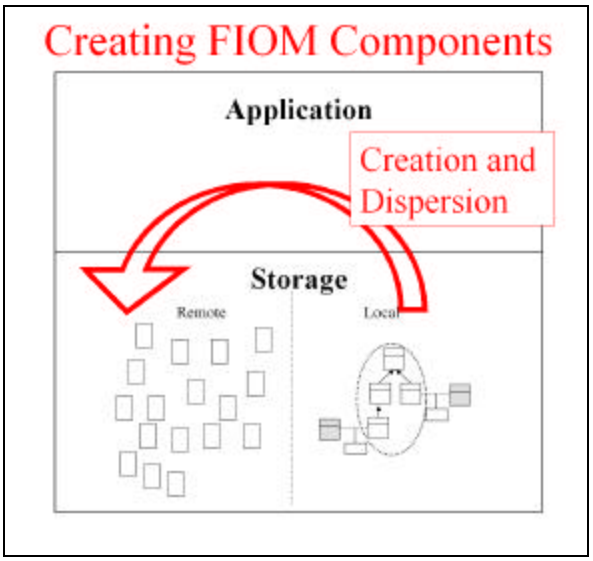

Figure III-14. Creating FIOM Components For Remote Storage.

In traditional files storage, a data file is opened, read and if changes are made, the new data is written to the file before closing, thus changing the file. The Local data structure is never directly written back to Remote, as the Local structure is analogous to a snapshot of the distributed data. In the traditional bi-directional method of file storage concurrency issues abound and so our read-only, one-way method is preferred. When new components are added to the Local data structure during FIOM construction, these changes must be saved persistently. To add the components to Remote storage, the individual components are created in accordance with the Remote data structure and then introduced directly to the Peer networking layer, which handles the distribution and persistent storage. The addition will be available to the other peers via their application when a new snapshot of Remote storage is collected and combined to form the Local storage data structure. For the heavy-weight components the main goal is to generalize or specialize a component, again without changing components already existing in

Remote storage, so that the read-only persistent storage is appropriate for this type of component as well.

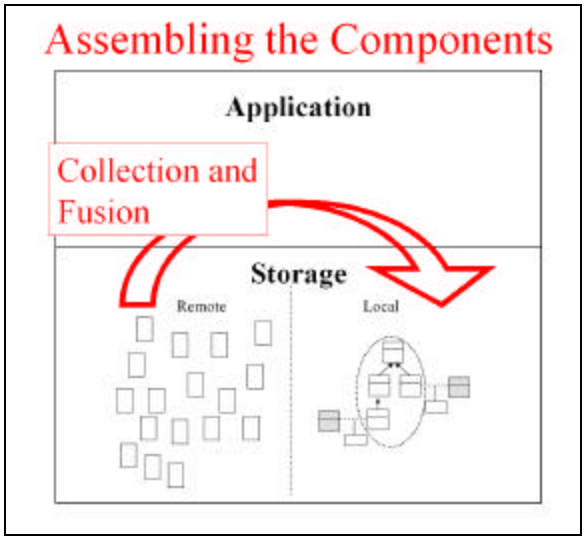

Figure III-15. Assembling The FIOM From Distributed Components.

The aggregation of Remote into Local also requires an application process as depicted in Figure III-15. In this case, a request sent on the peer network returns replies that provide Remote storage components of interest. The application then generates in memory the Local storage lattice by converting Remote storage components into Local storage format, and placing each component into the correct orientation in the lattice structure. The lattice is then accessible to the user by way of the application GUI. The user therefore works with an FIOM or perhaps even just a specified sub-set of an FIOM in a consolidated context, despite the underlying distribution of the persistent storage.

The architecture promotes the existence of multiple copies of light-weight components by storing copies of Remote components at each peer that relays or receives a component in reply to a Peer-Cloud request. For example, a peer requests a light-weight component and the component is then migrated to the requesting peer. The peer receives the component and stores a copy in its own locally held Remote storage, thereby caching or replicating the component for its own future use as well as for future use of the rest of the peer network.

The storage locations exist at the edges of the peer network because the participating peers each provide the appropriate interface and storage space for persistent storage. This is desirable, because there is no requirement for additional infrastructure, nor for centrally located servers or storage. From the peer perspective, the distributed storage appears to be encapsulated within the peer network layer, though in reality the Peer networking layer is only responsible for the messaging and transport functions. The light-weight and heavy-weight components must be stored on host systems and Distributed OOMI only references and accesses them in the context of the peer network. Therefore the Storage layer must be placed above the peer layer in the layered architecture stack. The Remote portion of the Storage layer is considered logically to represent the entire distributed storage available across the peer network. This distributed data structure fits the environment more naturally than other data storage architectures. Implementation is certainly more complex than a traditional client/server architecture with centralized mass storage, but the benefits, particularly the scalability of the structure are worth the investment. XML provides openness and promotes longevity of the persistently stored FIOM components, and fusion of the distributed data into a lattice makes the data usable at the application level. The issues associated with the concept of the Local storage lattice structure are unfortunately not trivial and so are the subject of the next section.

E. INHERITENCE AND THE LATTICE

In all things there exists hidden complexity and Distributed OOMI is not the exception. A future work item for Young's OOMI is the solution to the FEV inheritance hierarchy requirement, which is a simple concept with a very difficult implementation. Recall from Chapter II that we desire to relate FEVs contained within an FE in such a way as to allow direct type casting from one view to another view. The problem with implementing this type casting capability is that the hierarchy must be constructed in effect backwards, starting with the children and then creating the parents and other ancestors by generalizing the characteristics of two or more children. To add to the difficulty, additional children nodes will be added over time, thereby introducing a probability that the initial Generalization is not sufficient for all types of children nodes.

Object Oriented (OO) programming languages vary on their approach to inheritance. The majority of them do not directly support Generalization, adopting instead the idea that the most general object exists first, and that subsequent objects are more specific descendants of the initial parent object. With OOMI, there is no such presupposition that the most general FEV is introduced to the FIOM first. In fact, most FEVs will be very specific because of the requirement for their FCRs to be in one to one correspondence with at least one CCR. Distributed OOMI therefore faces a greater inheritance challenge than OO languages, because FEVs will be both Generalized and Specialized during construction of the FIOM. This is difficult to implement in a language such as Java that does not support Generalization with a native keyword or command. Because Distributed OOMI is implemented in Java, it seems appropriate to suggest a solution that allows both Generalization and Specialization of the FEV FCR Schemas. While Distributed OOMI does not offer an exact solution to this problem, the lattice approach suggested below meets the requirements of OOMI, fits within the distributed environment and allows both Specialization and Generalization of components of the inheritance hierarchy.

1. The Trouble with Trees

A simple and the most common concept for representing an inheritance hierarchy is a tree form, in which some node is the most general, and all other nodes are related along the branches of the tree according to the level of detail captured by the node. Most object-oriented languages use this concept in which the root is defined first, and all other nodes are extensions (specializations) of the root's general definition. In a distributed environment, how shall the most general node be defined? How shall levels of abstraction be determined? How many levels are required to be created to support the join of two FEVs? What should be done about multiple inheritance?

The answers to these questions may lead in the direction of requiring centralized definition of the majority of an FIOM inheritance tree structure, with the independent IEs adding only the leaf nodes that correspond to their particular system. If a centrally developed solution were feasible, the OOMI would not be necessary. It is not the case,

however, that DoD can define the entire Federation centrally and inform the system contractors where and how to connect their systems to the Federation.

Consider the following example in which three Views of a Federation Entity, containing FCR Schemas named respectively FCR1, FCR2 and FCR3, are to be organized into an inheritance hierarchy tree. FCR1 contains attributes *attr1, attr2* and *attr5*, FCR2 contains attributes *attr1, attr2* and *attr3* and FCR3 contains attributes *attr1, attr3* and *attr6*. As shown in Figure III-16 one possible top-down result involves a multiple inheritance relationship and three levels of abstraction.

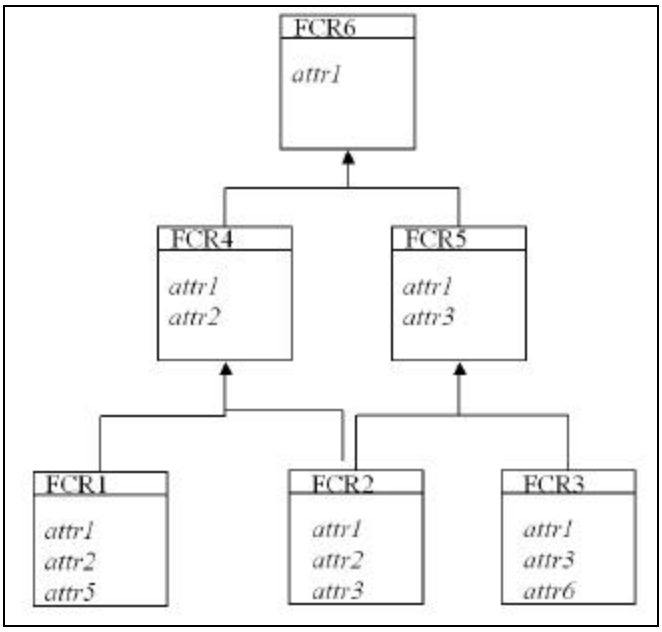

Figure III-16. FEV Inheritance Hierarchy In Tree Form.

The level containing FCR4 and FCR5 as well as the root node FCR6 are required to complete the inheritance hierarchy, but may represent a node that is not in one to one correspondence with a component system, meaning a CCR will never be associated with the node. The scope of the DoD interoperability problem requires a Federation that will encompass thousands of systems. A centralized and top-down approach would require prohibitive coordination between the specific system contractors for the target systems and so a distributed method of creating the FIOM hierarchy is desirable. By considering the FEV inheritance hierarchy to be a lattice, rather than a tree, Distributed OOMI allows

the hierarchy to be built bottom-up. This approach requires considerably less horizontal coordination, less central management and the issue of multiple inheritance is completely avoided.

Unfortunately, at the time the Federation is created, the entire structure of the Federation cannot be known. Top-down is therefore not a feasible approach for the construction of an FIOM. To take advantage of the native type casting of our implementation programming language, however, we would have to represent the inheritance hierarchy as a tree, defined essentially in a top-down fashion. The OOMI does not require a rooted inheritance hierarchy and so we suggest a lattice approach that allows bottom-up creation of the FIOM, despite the fact that native type casting will not be available for use in defining the hierarchy.

2. Lattice Concept

The term lattice is useful when describing the FEV inheritance hierarchy because it brings to mind an image of what is really happening when we relate FEVs to each other. From any given node in a lattice, there can exist paths to other nodes that are more general or less general than the current node with fewer restrictions than the tree model. The lattice simply captures these paths and provides the methodology for traversing the paths to develop and define relationships between nodes.

AN FIOM begins with a single FEV for any FE. This first FEV contains an FCR with attributes and methods that are in one to one correspondence with the attributes and methods of the first CCR. The next CCR to be registered requires a one to one correspondence with an FCR that may prompt the creation of a new FEV. The second FEV will relate in some manor to the first FEV by either a Generalization or a Specialization. To avoid a dependence on native type casting and a tree structure, the Generalization or Specialization is defined explicitly and external to the FEV objects. We term these external type casting mechanisms *transforms* and they behave in a similar manor to the translations that associate CCRs and FCRs. Transforms only work in one direction, however, and so we include both an *up-cast* and *down-cast* transform for every association that we desire to be bi-directional in the lattice path. By definition, the up-cast transform is a traversal of a lattice path that generalizes, and a down-cast is a

traversal that specializes. A condition of the structure of the FCR objects is that a generic up-cast and down-cast interface be defined that can be used when creating and accessing the transforms. This stipulation is paramount, because the generalization and/or specialization that require the up-cast and down-cast transforms will normally not be defined until after the FCR is written to persistent storage as a read-only Remote storage component.

In the following example, the CCR and FCR-to-CCR Translations are omitted to simplify the discussion and figures. The assumption of this example is that FCR1, FCR2 and FCR3 have been created as part of the registration process of three corresponding CCRs. The lattice shown in Figure III-17 first contained FCR1 as the initial FEV. The addition of a second FEV occurred sometime later, at which point FCR2 was associated with FCR1 via the transforms shown in the figure. In keeping with the definition, the up-cast transform defines the path from FCR2 to FCR1, with the down-cast transform defining the reverse path. The name of the path and relative direction is irrelevant after the path is established, because casting will occur via the interfaces defined as part of the FCR structure. The name of a transform is not as important as the fact that it is defined in accordance with the interface that enables its use.

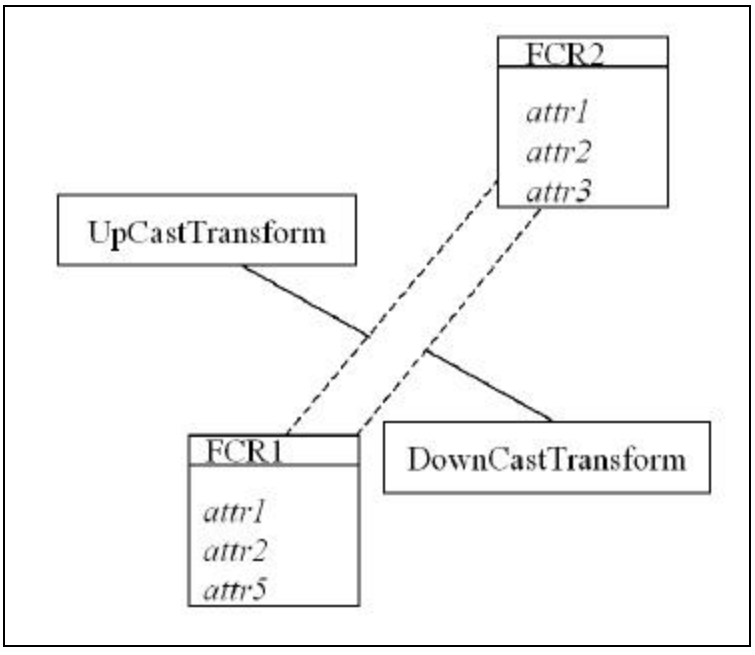

Figure III-17. Lattice example of Up and Down Casting transforms.

As more systems are added, the lattice expands without restricting where the next FEV can be connected. For example, when the third node is added to the example lattice, FCR3 can be connected either to FCR1 or FCR2. As discussed in Chapter II, the Component Model Correlator will be invoked to find the closest correlation between the attributes and methods of the third CCR to be added. In the case of our example, FCR2 contains both *attr1* and *attr3*, but lacks *attr6* and so FCR3 is created based on FCR2 by adding *attr6* and excluding *attr5*. Figure III-18 shows the result of adding the FEV containing FCR3. The hierarchy shown is equivalent to the inheritance hierarchy shown in Figure III-16 without including additional FEVs that contain FCR nodes that may not be useful as attachment points for CCRs.

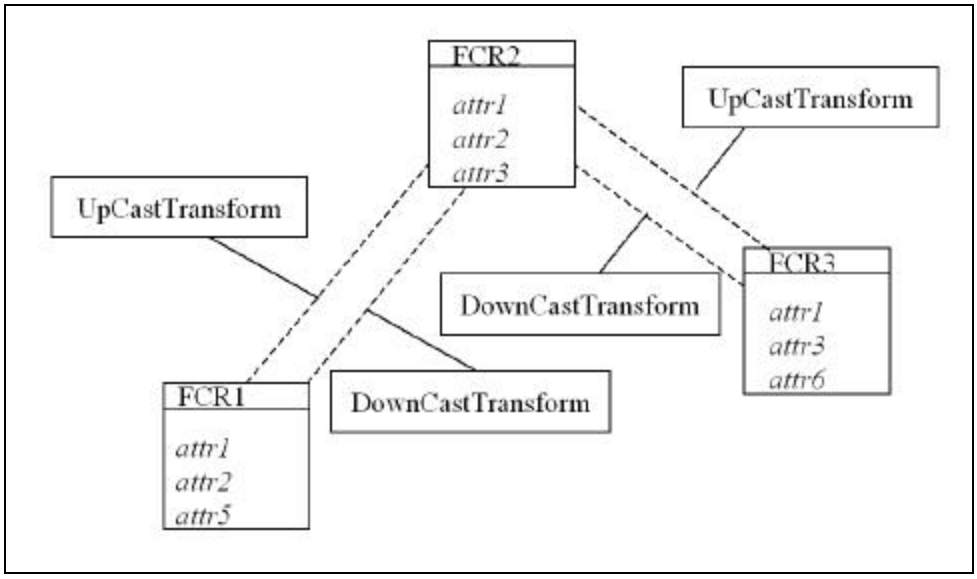

Figure III-18. Lattice With Third FCR Added.

In the distributed sense of an FIOM, the existence of a path from one FEV to another along the inheritance hierarchy is the valuable piece of information. Gathering the Remote components of an FIOM and using them to re-constitute the complete lattice structure for an FIOM is rarely required, because the complete lattice represents all possible connections between all the Component Systems. A partial FIOM Lattice is all that is required during the phase of the OOMI known as Translator Generation, during which a single path between two specific Component Systems is traversed. This phase is essentially independent from the generation of the individual FIOM components and so

even a partial FIOM lattice is a structure that is not required during every phase of FIOM construction. The chains of references relating FIOM components are stored in light-weight components and are used to search for the existence of paths between specific FIOM components. In the case of Translator Generation we search for the path that connects two CCRs because this represents the desired interoperation between the two target Component Systems. When the path is known, the light-weight components are again used to identify the required heavy-weight components, which are then collected and assembled into a partial Lattice. This approach minimizes the requirement to transmit and store locally any heavy-weight components that are not required for the specific phase of FIOM construction in question.

Another benefit of providing our own inheritance solution is the ability to include and exclude attributes and methods as required. Recall that FCR3 was based on FCR2, but is not directly an extension nor generalization of FCR2. This ability reduces complexity and avoids the creation of intermediary Federation components that fill the gaps between FEVs in the inheritance hierarchy. The lattice concept is not entirely defined by this research however, and not fully implemented. Chapter IV describes the level to which this concept is used in the current implementation of Distributed OOMI, but the complete design and description of the FIOM lattice concept is an open topic for future research.

F. DISTRIBUTED OOMI CHALLENGES

In Chapter II we introduced a common set of challenges that all distributed systems share and that provide the standards and measures by which a distributed system may be judged. Distributed OOMI addresses each of these challenges deliberately in order to mitigate weaknesses of non-distributed systems by leveraging the distinct advantages of the distributed environment. Because this architecture includes both a data structure and a layered architecture, some of the challenges are handled by the design of the data structure and others are handled in the way the data structure is accessed and used within the layered architecture. The following sections describe our approach to the challenges in greater detail and explain how the challenges are mitigated. We've already discussed the function of the layers of the Distributed OOMI architecture and now we

explain how the decisions made in developing the architecture relate to the challenges of distributed systems.

1. Heterogeneity Challenges

Heterogeneity can be dealt with in one of two ways, but the end result is always the creation of a homogeneous layer in which the application execution occurs. The direct approach builds this homogeneous layer by providing a specific software solution for each host system environment. Normally this implies a special build of a programming project so that it can execute in the target operating system environment. The other approach is the indirect method, which relies on some external solution already in existence, to provide the homogeneous level of abstraction. With Distributed OOMI we use the indirect method of resolving heterogeneity and choose XML as the primary serialization format for persistent storage.

There are two heterogeneity challenges that Distributed OOMI must overcome. The first is heterogeneity of the host systems participants will use to build the Distributed OOMI. This is partially addressable by choosing an implementation for the data structure that is portable across the majority of potential host platforms. We choose XML and Java for their portability. The other consideration with respect to heterogeneity is the network between the host systems participating in the construction of an FIOM. We use the lowest common denominator approach to this challenge, by attempting to minimize the physical size of distributed components that must traverse the network while at the same time using caching and multicasting to limit the number of times a component must be transferred.

The network over which the distributed applications collaborate to build the FIOM can have heterogeneous effects as well. Consider the vast potential for bandwidth restrictions on the links between the host systems in use to construct the FIOM. The architecture for the Distributed FIOM must therefore take into account the need to migrate the components of the model on the host network without assuming infinite capacity. As there are currently no quality-of-service (QoS) guarantees available across the entire transmission network, the only dependable solution is to assume that network resources are limited in capacity and will introduce considerable delay to packet

transmission. We combat this challenge by dividing a large and complex FIOM into a multitude of independent storage components each of which is much much smaller than the complete FIOM.

2. Openness Challenges

Openness is concerned with allowing participants to choose their own method of collaboration. XML and Java provide openness to distributed OOMI. Openness is important because the FIOM is desired to be a collaborative effort for construction and use. Adding to the FIOM requires Specialization and Generalization as the FIOM expands incrementally from an initial registration of two CCRs into a mass of components. In one sense the Federation is built on a first-come-first-serve basis and need never be any larger than that which is specifically required to provide translations between registered CCRs. Recall that if an FCR that is in one-to-one correspondence with a CCR cannot be found in the Federation, one must be created by either Generalizing or Specializing from an existing FCR. In order to allow this to occur, the existing FCR must be openly available for inspection to support the creation of a new component that can be associated via a simple up-cast and down-cast transform. A closed proprietary encoding of FCRs would not allow this vital function to take place, as the existing FCRs source encoding will need to be copied, modified and stored under a new name.

3. Security Challenges

The two types of security are secrecy and integrity and there can never be a complete solution to either or both. With that in mind, the secrecy challenge in general for Distributed FIOM is handled by making the existence of the Federation ambiguous. Because the FIOM only exists and is accessed within a P2P network, systems must participate as peers to be able to access and use the components of the FIOM. Using the best of currently available security solutions can strictly control access to the peer network. Within the Peer network, secrecy is not a concern, though integrity remains important. The distribution of duplicate FIOM components through the initial replication and then the caching of transferred components decrease the chances that all copies of a component could have their integrity compromised. The read-only nature of the

persistent storage provides additional assurance that integrity will be maintained. Again current security techniques are considered sufficient to mitigate this challenge.

4. Scalability Challenges

Scalability is achieved by creating a data structure that exhibits the property of being self-similar in all levels of the distributed data storage. Self-similarity allows consistent complexity to be applied across the Distributed FIOM data structure. The goal is to push processing requirements to the edge of the network and as high up the OSI stack as possible, thereby causing the least amount of requirements to be made of the supporting infrastructure. Additionally, the size and frequency of transmittal for Distributed FIOM components is deliberately organized to minimize peer network traffic. A Power Law describes the relationship between the number and relative size of light-weight and heavy-weight components. Though there are potentially very many communications involving light-weight components, the size of these components is very small. On the other hand, heavy-weight components are larger in size, but there are relatively few of them being transmitted. Additionally, the peer network uses caching in its implementation of multicast to avoid duplicating the transfer of an FIOM component more then once over a given link.

5. Failure Handling Challenges

Failure handling in technical terms will be handled by the reliable services offered by the peer environment. In the distributed storage sense, caching and limited replication of both light-weight and heavy-weight components ensures that every component exists in three or more places, such that a replicated copy of a component is always available. For persistence in time, the intention is to store the FIOM components related to a given system on the operational system, so that as long as the system is connected to the Internet and operational, its registered Federation components will be available. This eliminates the need to maintain a dedicated storage network for the Distributed FIOM components.

6. Concurrency Challenges

Concurrency is handled by making the FIOM essentially write once/read only there after. When a component is written to distributed storage as part of the Distributed

FIOM, it need never be edited, updated or changed in any way and it will persist indefinitely in the FIOM peer network. When a change is required, a new component is created and sent to distributed storage, connected to the FIOM in the same manor as the first, only this time providing a new path possibility in the Distributed FIOM data structure. Collection and combination of the distributed components into the local FIOM makes the local sense of an FIOM different from the distributed storage and so concurrency here is not an issue, as the storage portion is only read and never written. In fact, the only writes to the Distributed FIOM are new components entering Remote storage for the first time. All other storage transactions are read or copy actions that do not modify content of the components. The lack of a possibility for simultaneous write therefore mitigates the concurrency challenge.

7. Transparency Challenges

Mitigating the challenges of distributed systems is not the only requirement. The solution must also preserve transparency in several respects. This section describes how the various components of the Distributed FIOM architecture relate to transparency requirements.

Location transparency for the OOMI involves hiding the fact that we have different methods for local and remote storage. Layers of abstraction in the Distributed OOMI implementation allow the partitioning of the storage solution by using two different yet compatible data structures. Normal read and write capability are built into the layers of the tool, although the actual behavior of these operations is different than typically discussed in operating systems.

Concurrency transparency is not an issue for Distributed OOMI, because there is no editing of persistent data. Multiple copies of read-only files exist and any changes or additions occur by way of introducing new file objects. The only concurrency issue involves a unique naming requirement; however, the inclusion of an Ontology as part of the OOMI eases this requirement. One problem that remains involves the possibility that two independent sites could potentially register non-identical components under the same name, at the same or nearly the same time. Thus two components with the same name would be available, with ostensibly different results depending on which one is used.

This situation is avoidable by implementing a simple process within the Federation Ontology Manager that prevents the independent introduction of new entries to the Federation Ontology. As the Federation Ontology remains an aspect of the OOMI open for future research, we defer further discussion of this concurrency transparency issue.

Replication is handled at the distributed layer via caching and some explicit copying at the time of first submission to the peers. Upon registration of a new component, it is sensible to assume that several copies should be dispersed about the network for robustness in the near term. However, as time goes on and components are requested and thus transmitted throughout the network, it is equally sensible to assume that transmitted components will be cached at intermediate peers in order to reduce future transmissions and further replicating the instance of the distributed components storage.

The failure handling and mobility transparencies are impacted by the implementation and cannot be captured completely by the architecture. We make suggestions and provide ideas as part of the prototype described in Chapter IV. The multiple copies of Federation components populating Remote storage and the Peer-Cloud approach to the distributed collaboration contribute to enabling these transparencies, though.

Performance transparency is addressed by minimizing the demands placed on the network infrastructure that connects the distributed participants collaborating in an FIOM construction effort. Because network resources, specifically bandwidth restrictions on some network links, may introduce delay we define network communications as reliable, but asynchronous. Additionally, we minimize the size of Remote storage components in order to avoid exceeding bandwidth restrictions of the underlying network. Lastly, the collaboration effort is intended to be distributed in space and time so that the requirement for participant communication and interaction with the Peer-Cloud is minimized in duration and frequency.

Transparency of scale is made possible through the use of two different data structures for local and remote storage and the peer network overlay. Consider that Local storage need not contain everything that can be retrieved from Remote storage. Locally,

the application user needs only those components applicable to the section of the FIOM on which construction is proceeding. Other portions can be left in Remote storage and requested when needed. Local storage can be purged at any time, as it is always re-constructible from Remote. The Local view is always a cohesive lattice and the user never really knows that many distributed components combine to form the Local view.

G. AGGREGATE ARCHITECTURE AND CLASSIFICATION

Putting the pieces together yields the complete architecture as shown in Figure III-19. Distributed OOMI operates on top of the existing Internet infrastructure. The Peer Networking layer implements a Peer Cloud, thus providing a distributed network environment based on the context of the collaborative construction of an FIOM. The Storage layer provides two data structures, one supporting distributed storage of Federation components as XML formatted text files and the other providing the ability to assemble the distributed components into a useful memory resident lattice of Java objects accessible via an OOMI IDE application.

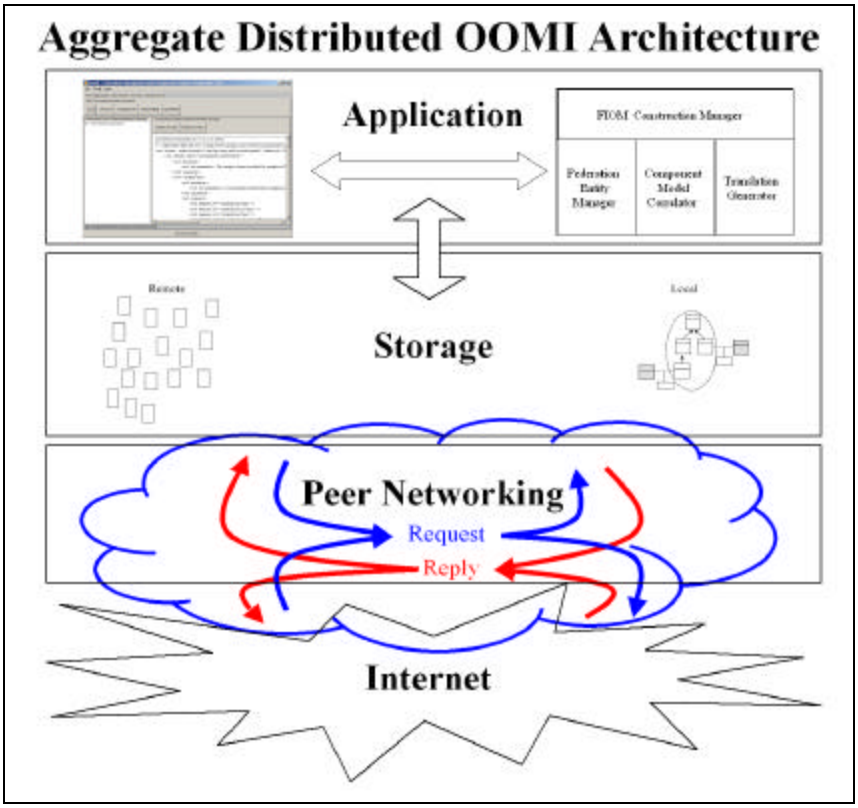

Figure III-19. Aggregate Distributed OOMI Architecture.

The application layer is the final layer of the architecture to be described. Recall from Chapter two that the OOMI IDE application is the product of several research efforts that focused on various functional capabilities required for the OOMI. The results of the previous work on the OOMI IDE are incorporated into Distributed OOMI at the Application and Storage Layers of the architecture. Distributed OOMI is an architecture, and so it is necessary to disassociate the application from the data. In the previous OOMI IDE prototype there was not so clear a distinction and a dividing line is therefore drawn between the data storage and data manipulation function of the OOMI IDE code used in the example Distributed OOMI implementation discussed in Chapter IV.

The Layered architecture is intended to promote a modular approach to implementation and competition in the form of independent yet interchangeable software solutions to each layer. For instance a desirable result would be for existing software engineering tool suites to incorporate the application layer interface for Distributed OOMI so that the existing application could participate in a Distributed OOMI collaboration effort. Similarly, until a complete implementation of a Peer-Cloud solution is available, existing distributed networking solutions are used to approximate the Peer-Cloud behavior. The Storage layer may be approximated, but the concept of dividing storage into Remote and Local is paramount to the architecture and should be preserved regardless of implementation approach.

Considered against other distributed systems in terms of the Classification Paradigms for Distributed Systems described in Chapter II, Distributed OOMI is a hybrid solution that combines aspects of the Implementation and Processor/Storage centric paradigms. The layered architecture and object-oriented approach contribute the characteristics of the Implementation centric paradigm. The self-similar distributed storage of the FIOM components and the Peer-Cloud abstraction of the participating systems make up the Processor/Storage centric contribution. Figure III-20 shows the relative placement of the Distributed OOMI ellipse. Notice that the classification ellipse for Distributed OOMI does not intersect the User centric paradigm. Though this research contains methods and ideas that could be useful in a User centric distributed system, Distributed OOMI is not intended to allow a single user to leverage the capability of the

distributed system. Instead we intend for multiple users to work essentially independently while collaborating within a common context.

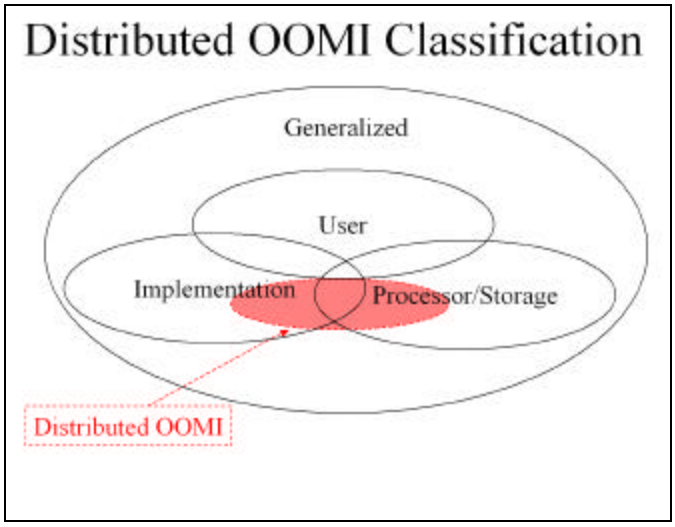

Figure III-20. Classification Of Distributed OOMI.

How Distributed OOMI may or may not be used in the future is uncertain, but the conceptual view of its functional deployment is described for completeness. The independently contracted organizations responsible for the DoD systems targeted for participation in an FIOM enabled interoperability effort will build the part of the FIOM corresponding to their respective systems. This means that each Program Management Office will work with system contractors to provide, via an Interoperability Engineer, the CCRs for the Program's system. The CCRs will be registered with an FIOM constructed in a collaborative fashion using the Distributed OOMI architecture. We hope to allow this process to occur on the Program's own terms, and so the modularity and openness of the architecture will hopefully encourage development of several Distributed OOMI enabled software tools. The collaboration between multiple Program Management Offices would take place via a peer network established specifically for the FIOM being constructed. The FIOM construction will occur over time, bottom-up, in a first-come first-serve manor, but Translators could be generated after just two systems have been registered.

In the next chapter, we discuss how Distributed OOMI is implemented in Java and XML and present an example of the process of creating an FIOM using Distributed OOMI. The decision to implement using XML and Java seems premature considering we are describing an architecture, but the OOMI itself is based on XML and Data Binding of an XML Schema to a Java object. Despite the possibility that this architecture could perhaps be implemented using alternative techniques and technology, we accept the assumptions of the OOMI and desire the consistency obtained by choosing to continue with techniques and programming languages in use. XML and Java are therefore intimately related to this architecture and at the same time support the distributed computing goals described in the previous sections of this Chapter.

IV. IMPLEMENTING THE ARCHITECTURE

A. BUILDING AN OOMI IDE

A primary tenant of the OOMI (described in Chapter II) is the use of computer automation to provide assistance in the creation of an FIOM. The work of [Chr01] introduced an example OOMI IDE, which has been expanded incrementally to incorporate the work of [Lee02] and [She02]. The collaboration architecture introduced in Chapter III provides additional guidance for the development of a prototype IDE. In some cases a refactoring of the original notion of the IDE is required to incorporate the Distributed OOMI data structures.

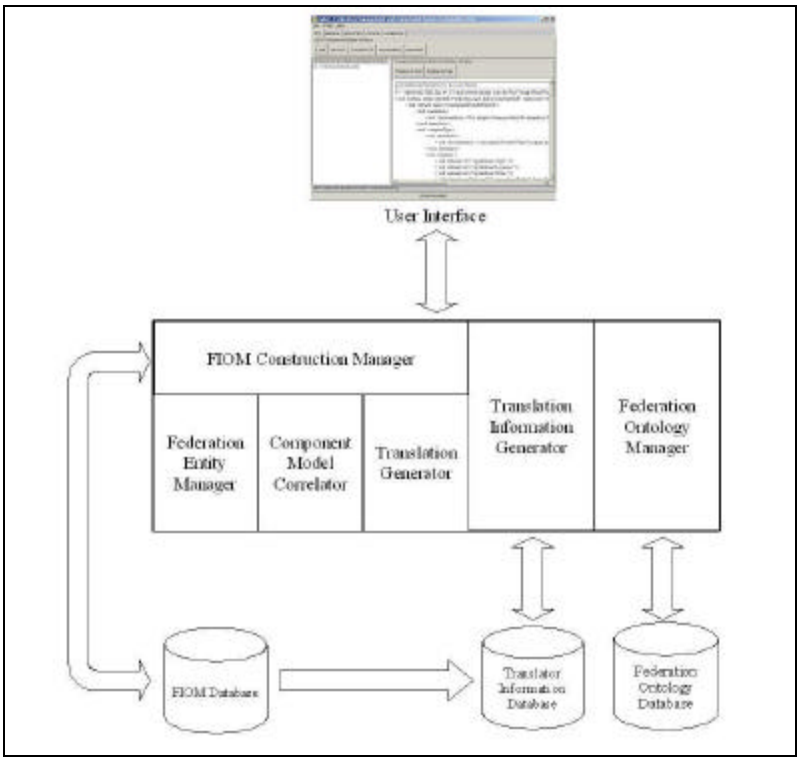

Figure IV-1. OOMI IDE Components. [from You02]

Notionally, any OOMI IDE is made up of a user interface, application code and persistent storage. To date the prototype implementation is modeled after the conceptual diagram shown in Figure IV-1. We present in this Chapter our experiences in creating the prototype and our view of the anticipated refactoring of the traditional OOMI IDE

component relationships to facilitate a prototype under the Distributed OOMI architecture.

1. One Big Application

The Distributed OOMI architecture refers to several layers, but the reality of the situation is that each layer is constrained by the assumptions of the layer below. The necessity to leverage the existing Internet Protocol (IP) stack causes the Peer Networking layer to occupy the application layer of the IP stack. The Storage and Application layers of our architecture also must be implemented in the application layer of the IP stack. Placing Distributed OOMI on top of IP means that all of the layers from Chapter III are contained within the application layer of the IP stack. Without a development effort to re-implement the underlying host operating system and the IP stack, there is little else that can be done. The prototype examples used to describe the Distributed OOMI implementation are all Java classes running in user space on the host systems as an application. The behavior is intended to be that expected from an operating system environment in which storage and peer networking functions are OS layers, executing in system space on the host platform. We make the distinction because despite the fact that the prototype is one big application, we attempt to preserve the modularity assumptions and the use of interfaces that are required for a layered implementation. As justification consider that in the future, should peer networking be available from the OS, it would be preferable to re-use the Storage and Application layer implementations on top of the OS provided peer networking, rather than starting from scratch. Alternately, the Storage and Peer Networking layers may be included under a different Application layer environment, such as an existing software engineering tool suite.

2. Implementation Language

We use the Java programming language for its cross platform capabilities and availability of existing Application Programmer Interface (API) re-usable code sets that simplify IDE construction. The complete list of APIs from the Java programming language developers' community is vast and will not be covered, but we will mention a few that shape the prototype implementation. The *Swing* API[Sun01] is used for the prototype Graphical User Interface (GUI) and provides an environment that makes use of

the window-based look-and-feel commonly in use at present. The Castor project's API[Bla01] is used for all data binding. Castor bundles several other open APIs, such as an XML Parser, and these resources may be used instead of separately acquiring and incorporating the individual open APIs. The difficulty with this approach is that the open APIs included in the Castor bundle may not be the most recent product available from the developer, or might not be the API that the implementation team is familiar with. For example, the original prototype used the Simple API for XML (SAX)[Chr01, HCD01] to parse and manipulate XML documents. For a brief period after SAX, the prototype used the JDOM open API until we realized that despite the simplicity of JDOM, we still lacked an important capability for XML parsing. Currently, Xerces J[Chr01, AAC+01] is the XML API of choice for the prototype; however a point we would like emphasize is that when using products from open development initiatives, flexibility and the potential for abandoning an API for an upgrade are a recurring part of the development process. By using the bundled APIs, the development effort depends on the organization creating the bundle to provide a product that meets all of our requirements, which is unfortunately unlikely.

The wide availability of open APIs make Java the obvious choice for development of a prototype in an educational setting. It seems natural to expect that Java would also be used to implement a deployable OOMI IDE application, because of the familiarity of the Java APIs, as well as the potential realization that technologies such as data binding are not maturing as rapidly when implemented in other programming languages. For the OOMI, any object-oriented language will do, and certainly anything implemented in Java could be implemented in C++, for example. For our research, funding and time were constraining factors and so the affordability, applicability and ease-of-use provided by the open APIs was the obvious choice.

3. Object-Oriented Style

The Object-Oriented Method for Interoperability obviously depends on using an object-oriented approach, but does the requirement for an object perspective absolutely require an object-oriented approach when implementing an IDE? We think that it does in order to maintain modularity and flexibility within the layers. The difficulty presents

itself in deciding what is the data for the IDE, what are the operations on the data and how should the objects be defined so that the right data is available at the appropriate level of abstraction throughout the IDE.

Our general approach is to consider each of the components in Figure IV-1 as an object, and to define the interaction between the objects with interfaces. Interfaces were implemented as adapters with stubbed methods to allow incremental development of modules independent of the base application code. This approach works well and introduces a concern that is at the center of the object-oriented paradigm. Considering each module of the IDE as an object means that each module therefore abstracts some data structure, which is central to the purpose of the module. To accomplish the modular design, however, some objects must be passed as parameters via the interfaces in order to communicate and collaborate between modules. The challenge in the IDE is defining both the shared and protected data so that each may be placed in the appropriate position within the container relationships of the application modules. Out of this realization came the notion that there exist objects that must be preserved as well as objects that are disposable. For example, the module of the IDE that interacts with the persistent FIOM storage must preserve the objects it contains, as these objects represent the components of the FIOM. Another module, such as the Translation Generator, depends on objects passed in via its interface to perform its intended function; however, the objects it receives can be copies of the originals, and thus discarded after being used. The GUI component of the IDE is perhaps the best example of this, as every object sent to the GUI for display is ultimately discarded.

4. Interfaces and Listeners

The cavalier attitude (advanced by the previous section) toward the shared objects that move between modules introduces a problem; particularly with the GUI, of how to deal with the inevitable case of an object that must be both protected and shared? By discarding shared objects, we create a single direction communication. To enable communication in the reverse direction, without waiting on an object to be returned, we use either an established interface or a listener construct. Technically, interfaces and

listeners both return an object, but the difference is that program execution is not explicitly halted while waiting for the return object.

The prototype IDE was developed ahead of the functional modules to provide an environment in which completed functional modules could be integrated and demonstrated, thus alleviating the module developers from the responsibility of generating a specific demonstration environment. The problem we encountered with using interfaces to establish the reverse communication path (again, particularly concerning the GUI) was that as modules were incrementally developed and integrated, the interfaces between the modules had to change to accommodate specific requirements of the functional modules that came to light after the GUI environment was created. The solution is a compromise in which the IDE level communications are conducted via the interfaces that separate the modules. The communications requirements of the objects introduced to the environment by the modules are handled by listener constructs.

For example, an IDE level requirement is the ability to display an FCR and a CCR side by side so that the Interoperability Engineer (IE) can choose the attribute and operation matchings required during Translation Generation [Lee02]. The Translation Generation module level requirement is to capture from the GUI the result of the IE's matching efforts. Because the behavior or methodology of the Translation Generation module is free to change over time, we desire not to capture in the IDE level interfaces any representation of a module generated object that would either limit module development or require changes to the underlying IDE structure, especially the inter-module interfaces. Lee accomplished this with the use of a listener that allows the results of the IE attribute matching to be sent directly to the Translation Generation module [Lee02].

The basic assumption made is that using generic Java Swing components for the user display with behavior handlers attached to the underlying code, allows for the most portable arrangement while providing specific solutions to OOMI IDE functional requirements. Listeners are preferred to interface method calls to simplify message passing for module specific functions; however, interfaces are important for general IDE functions.

B. ORIGINAL OOMI IDE COMPONENTS

Implementing the Distributed OOMI architecture changes several of the assumptions of the original OOMI IDE. We discuss in this section the state of the prototype IDE prior to the research into the distributed solution, so that we may introduce the distributed implementation as a refactoring of the current prototype.

1. The User Interface

The functional requirements of FIOM generation provide guidance for GUI form and function. These requirements are laid out by Young as the process for FIOM construction [You02]. Though the GUI should be disassociated from the functional application code, it must present the features and terminology that are meaningful to the OOMI processes in order to establish a context that supports FIOM creation.

Explicitly designing a specific GUI was determined to be the most meaningful way to demonstrate OOMI concepts, though in reality, OOMI processes will likely be included as part of existing software engineering tool sets. The organizations responsible for the component systems to be registered in an FIOM most likely already have organizational software processes and automation assistance tools in place that manage software engineering and component system development. The most beneficial solution for these organizations is an FIOM construction solution that fits into their existing software engineering environment. To facilitate this, the specific OOMI User Interface (UI) functions would be imported to the existing environment package in the form of a plug-in, while preserving the attachments to the underlying OOMI application functionality demonstrated by the prototype modules.

The current prototype GUI provides for the OOMI module developer the ability to demonstrate functionality in an environment similar to what an actual implementation may provide. The development of more than the most basic GUI functions is considered to be unnecessary in light of the argument presented above. The challenge for the GUI is determining the boundary between nice-to-have features and the OOMI IDE functional requirements. Understandably, this boundary realistically is established between the capability to be demonstrated and the complexity of the implementation. For example,

GUI features such as the ability to 'Undo' and 'Redo' user commands are nice to have, but the complexity of implementation out weighs their benefit in the prototype IDE's ability to demonstrate FIOM construction concepts. Alternatively, the ability to display FIOM objects using a Unified Modeling Language (UML) representation within the GUI is a complex implementation that adds to the demonstration of OOMI concepts.

2. The Construction Manager

The prototype IDE designates this module as the central component in the application. All other modules plug into the IDE via the Construction Manager. This section of the IDE handles user events passed via the interface with the GUI and redirects user requests to the appropriate sub-module. As the director for the IDE activity, the Construction Manager must maintain a view of the UI, provide feedback to the user and make available to sub-modules the appropriate data. While it's not necessary to follow every motion of the user input device, we must be able to determine the context within which the user selects a given action. By understanding what the user sees and what the options are, we can store key bits of information that provide context for complex operations. One such storage construct maintains references to the CurrentCCR and CurrentFCR, which simply keeps track of which CCR and FCR the user is focused on. The construction of the FIOM requires pairing a CCR with an FCR and so it is natural that a user would focus on one of each component, and then desire to perform a function using the two components currently in user focus.

Many user requests require an aggregation of sub-module functions, and the Construction Manager methods contain the logic for multi-step tasks. As the manager of more complex combinations of sub-module functions, error checking and user feedback are necessary functions of the Construction Manager. For example, to invoke the Translation Generator, the Construction Manager must know which CCR and FCR occupy the user's focus and retrieve these from the data store. These components are then passed to the Translation Generation module to create the GUI component that can be passed to the display for user interaction. The Construction Manager is responsible for handling the occurrence of an error or failure during the sequence of steps.

In some cases the Construction Manager must develop the Swing component that will be displayed to the user. Some generation of Swing components is natural for the first layer beneath the GUI, but below the Construction Manager, the dependence on implementing a GUI Swing component should include only those situations when module specific functionality not available via the inter-module interfaces is required. Refactoring the sub-modules to bubble the Swing components up to the Construction Manager level is preferred, in order to keep UI and application code separate. While in the refactoring the Construction Manager becomes more closely tied to the GUI, the sub-modules become less connected with a particular UI implementation.

3. The Federation Entity Manager and FIOM Database

The application program logic for creating and manipulating FIOM components within the IDE is partially provided by the Federation Entity Manager. Probably the most important function of this module is the Castor enabled Java source generation from XML Schema documents. The resulting source code must be compiled and then class objects instantiated; all are functions supported by this module.

The FIOM Database was intended to provide a single module through which access to the persistent storage on the host hard drive could be accomplished. In this capacity, the module was also intended to contain and protect the data structure of the FIOM within the IDE. The shortcomings of this implementation model became most apparent when the Federation Entity Manager needed to access the hard drive to retrieve XML Schema files and the concept of sharing an FIOM grew from sharing simply within the IDE to sharing between multiple IDEs. While the notion of abstracting system access and containing it in a single module is attractive, the usefulness within the prototype IDE is not commensurate with the complexity of the implementation. The idea of isolating and protecting the FIOM data also seemed at the time to be valid, but the complexity of passing sub-sets of the data to the other modules via various indirections is unnecessarily programming intensive.

C. REFACTORING FOR DISTRIBUTION

Considering the construction of an FIOM in the context of the Distributed OOMI architecture clears up some implementation concerns from the original prototype IDE,

while introducing new challenges. A distributed implementation uses layering to disperse the complexity according to various levels of abstraction. The perspective of the layer in question determines the level of abstraction in which the data is viewed, and the concept of a monolithic data set existing external to the majority of the IDE modules is abandoned in favor of a logical concept of the data maintained by several data models specific to a particulat layer that do not necessarily individually represent the entire data concept in a single location. For example, the Construction Manager module does not need to store or handle actual FIOM components to perform its function; however, it must, in some fashion, represent the existence of the components so that a context is established in which the user can designate what to do, when to do it, and which components should be involved. The user may only need the names or graphical representations of the components to be able to communicate to the Construction Manager the desired action. The data used by the Construction Manager then is potentially very different from the data required by another module, and should be considered independently.

Interestingly, the idea that each layer and each module within a layer, for a particular implementation, requires a specific view of data that is representative of a larger concept reduces to an interoperability problem that can be solved using the OOMI! In this case, the modules of the IDE implementation are considered component systems in an FIOM that resolves the differences in view and representation. When programming a single application, introducing the complexity of an FIOM to define the interaction between program modules is likely prohibitive, but it is conceptually feasible. For our purposes, we'll simply discuss what the data requirements are for each layer and suggest how the modules in a distributed application might abstract the overall concept captured by the data. An example inter-module requirement is a method of connecting to the Storage Layer data structure that can extract an abstraction of the current FIOM and map the result into the Construction Manager data structure. It is potentially possible for implementation modules occupying the same layer to use a single data abstraction, but equally possible that all modules of a layer could have an independent data representation in order to capture the data abstraction at the appropriate level.

1. Distributed OOMI Application Layer

Originally considered to represent the portion of a computer program that contains the functional logic while being expressly separate from the data, we now see that this layer is not characterized by an underlying split between data and application functions, but rather by a shift in context surrounding the data. Implementing this layer requires data structures that are perhaps initially populated with data from a lower layer, but over time, the application layer becomes populated with data provided via interaction with the user, as well. User interaction is equally important in establishing the context in which the data is ultimately manipulated.

An IE is principally concerned with only a small subset of the total FIOM. The act of registering a component system CCR involves in essence a single FCR and a single CCR. Until the Translation Generation step of the OOMI is reached, the IE requires only a high-level view of the FIOM. We can provide this view in the form of a tree representing the containment relationships of the FIOM structure. When working with particular heavy-weight FIOM components, a UML representation is a better representation than a tree. Finally, the inheritance hierarchy when considered a lattice, is difficult to represent with a tree structure, and may be prohibitively large to display with UML. The goal of the data structures of the application layer should be to establish the context of the user interaction as accurately as possible. With a data structure defined that meets the requirements of a module or layer, the integration of that data structure with the other data structures can be solved via direct mapping, or potentially via a more general means, such as an interoperability model.

In a distributed implementation, the application still consists of the GUI and the functional modules identified for the original OOMI IDE. The Construction Manager and GUI maintain a similar relationship, except that we expect a tree based data structure to be included at this level to allow for presentation of the FIOM at a high level of abstraction as well as track the user's focus. The tree data structure is populated based on the containment relationships as captured by the light-weight FIOM components. The Construction Manager will use the Federation Entity Manager sub-module to collect the FIOM components and build the tree. User interaction will start the process, most likely

by identifying, by name, an FIOM to work with or by creating a new FIOM. The presentation of heavy-weight components to the user requires an additional data structure. And finally, to present the FIOM Lattice to the user via the GUI, the Construction Manager will require a separate data structure that contains an abstraction of the FIOM Lattice. The form of this structure is not yet decided, but the population of the structure would be facilitated via the Federation Entity Manager, likely coincidental with the construction of the FIOM Lattice.

The Federation Entity Manager will provide access and handling for the light-weight and heavy-weight components of the FIOM, including the creation of new components. As with the original IDE implementation, the Castor-enabled data binding process is conducted by the Federation Entity Manager module. A new functional responsibility of this module is the assembly of an FIOM Lattice in Local storage. This process involves instantiating instances of heavy-weight components from Remote storage and populating a lattice data structure in Local storage. When necessary, new FIOM components are created with functionality provided by this module, and then written to persistent storage as XML files in the Remote portion of the Storage layer.

2. Distributed OOMI Storage Layer

The purpose of the Storage layer is persistence of the FIOM and interaction with a portion of the inheritance hierarchy. In a way, this layer logically represents a monolithic data store by responding to Application layer requests concerning the entire FIOM. The difference between our anticipated implementation of the Storage layer and a monolithic data source is the possibility in our architecture to consult the underlying peer network. Also, we stipulate in the architecture that the Storage layer requires Application layer interaction to create the Local storage data structure. The inheritance hierarchy constructed in Local storage is therefore bounded by Application layer context, making the resulting lattice both non-monolithic and perhaps even a unique construct, due to the non-persistence of the Local storage structure.

The two types of Storage layer Remote *storage components* are the light-weight and heavy-weight components of the FIOM discussed in Section III.D.4. Data binding is the anticipated method for producing class instances from persistent storage in XML

formatted files for storage components of both types. Implementation of the creation of new storage components is possible in one of two ways. First, a new class instance could be generated, configured and then marshaled to XML for persistence. The second method is to generate the new component as an XML document, configure the component and then write out the document to persistent storage as a cohesive unit. To facilitate the first method, a generic class with a defined marshal and unmarshal method must exist for every component to be created in this fashion. Both FIOM component creation approaches are useful and recommended. Light-weight components can be created easily and efficiently by instantiating classes that are marshaled to XML, while heavy-weight components are more naturally created by generating the class from an XML Schema. For example, the normal method under the OOMI for creating a heavy-weight CCR component is to use Castor to generate the class directly from an XML Schema.

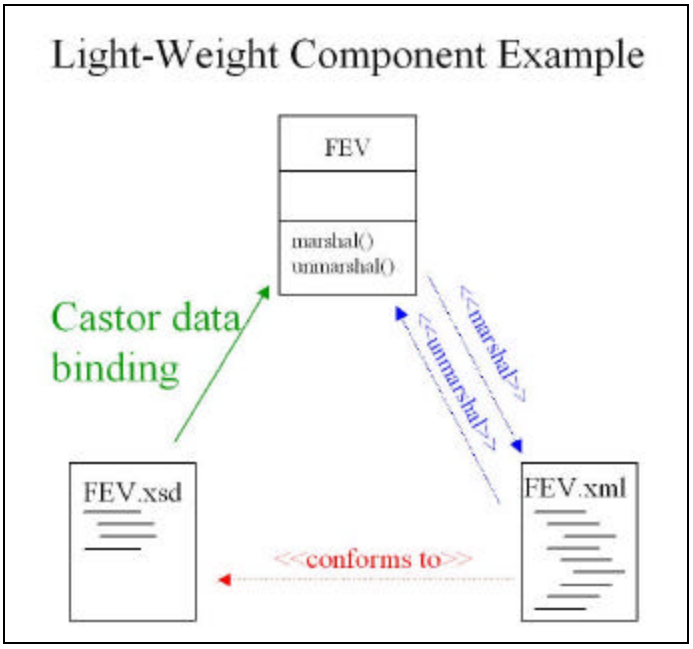

Figure IV-2. Example Light-Weight Component Relationships.

The structure of the light-weight components is reused; meaning the Remote storage representation for a light-weight component is an XML document, rather than an XML Schema. The generic light-weight class that contains the marshal method used to

create the XML document is created initially from an XML Schema, however. Figure IV-2 depicts the relationships between artifacts involved in the creation and use of an FIOM light-weight component for Federation Entity View (FEV) creation and use. Example XML Schema for all light-weight FIOM components are included in Appendix A.

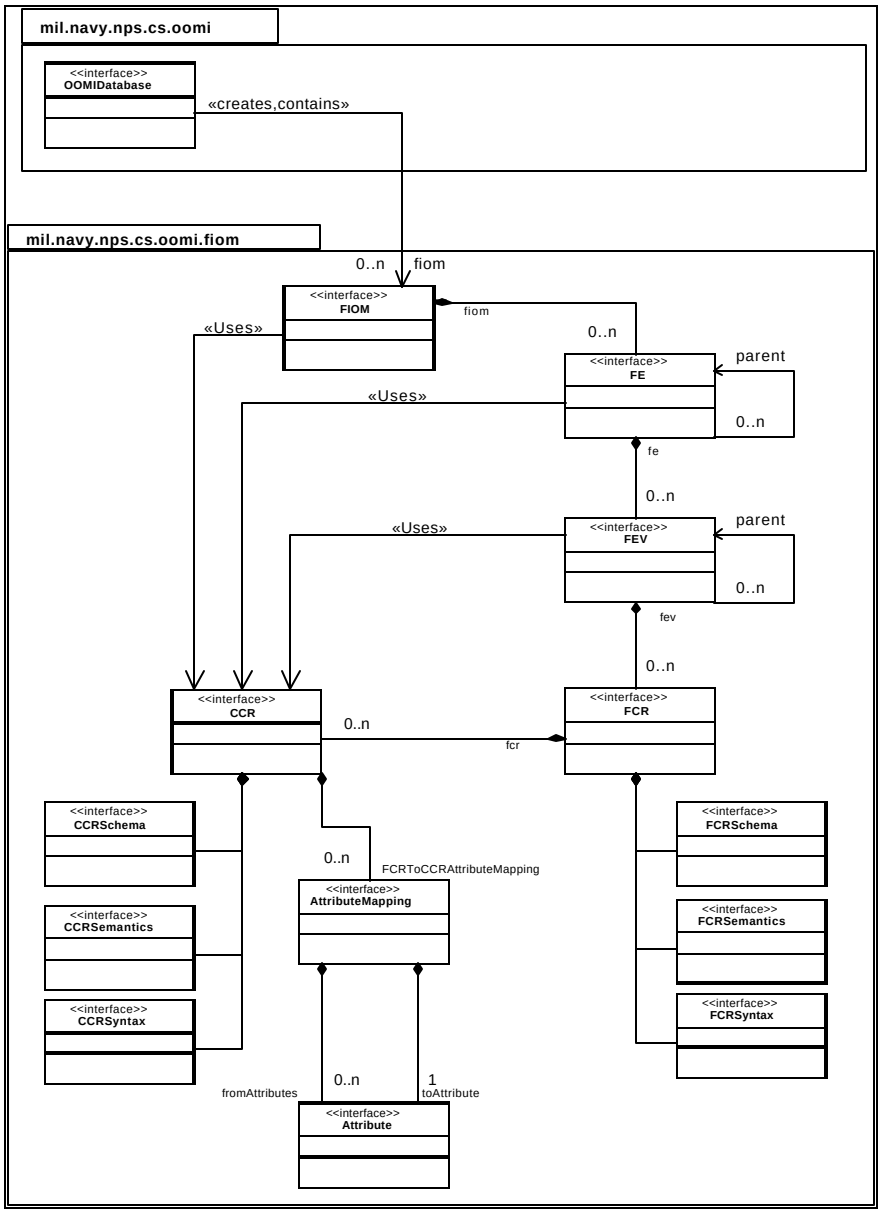

Figure IV-3. Packages: mil.navy.nps.cs.oomi And mil.navy.nps.cs.oomi.fiom. [from Lee02]

In the original FIOM data structure presented by [Lee02] (see Figure IV-3), the data binding process created a CCR Schema or FCR Schema that extended, respectively, a CCR or FCR predefined class. The CCR super class provides additional data storage capability and functionality that assist in the creation of CCR-to-FCR translation classes. Additionally, the organization of Lee's FIOM data structure containment causes the CCR to be contained within the FCR, rather than being contained within the FEV that defines the relationship between the CCR and FCR, as we suggest in Chapter III.

Despite the desire to re-use Lee's structure to represent the FIOM Lattice in Local storage, these differences between the intent of the architecture and the implementation of the structure need to be resolved. Changing the structure of the CCR and FCR super classes to reflect the intent of the architecture and facilitate the characteristics of the FIOM Lattice could cause the loss of the attribute mapping data storage provided by the CCR super class. This data storage is required for the operation of the Translation Generation module also provided by [Lee02]. One possibility is to continue to use the concept of data binding into a specialization of a CCR super class that supports the architecture requirements, while attempting to preserve the original attribute mapping capability. Ultimately, Lee's structure does not support the Lattice concept, because it models the FIOM according to the logical containment, or tree, view. To traverse an FIOM Lattice path from source CCR to destination CCR using a tree model of the FIOM requires descending multiple branches of the logical tree view to find the heavy-weight components. The complexity and overhead of finding the Lattice paths in a tree representation of an FIOM is excessive, and so we recommend pursuing a lattice-based data structure for Local storage of the FIOM. As mentioned earlier, each module of the Application layer may benefit by having its own data structure, suited to the functions of the module in question. We therefore also recommend that Lee's data structure be implemented within the Translation Generation module to serve the needs of Translation Generation, and that a means of populating this data structure be provided for via the Federation Entity Manager.

How then should an FIOM Lattice be represented in Local storage and how might such a structure work? The lattice consists primarily of two constructs; one that contains

the components that model a particular view of the real-world entity, and another that represents a generalization and specialization relationships between views. Together these self-similar generic lattice constructs allow for dynamic organization of innumerable lattice instances, of various sizes, while preserving an abstract notion of overall lattice behavior.

The first construct is based on the OOMI defined FEV (Section II.A) and provides a container for an FCR and zero or more CCRs. Each CCR is associated with the FCR via an FCR-to-CCR Translation class. Figure IV-4 depicts the OOMI relationship between FEV components and suggests the appropriate generic container relationships using superimposed circles and dotted lines.

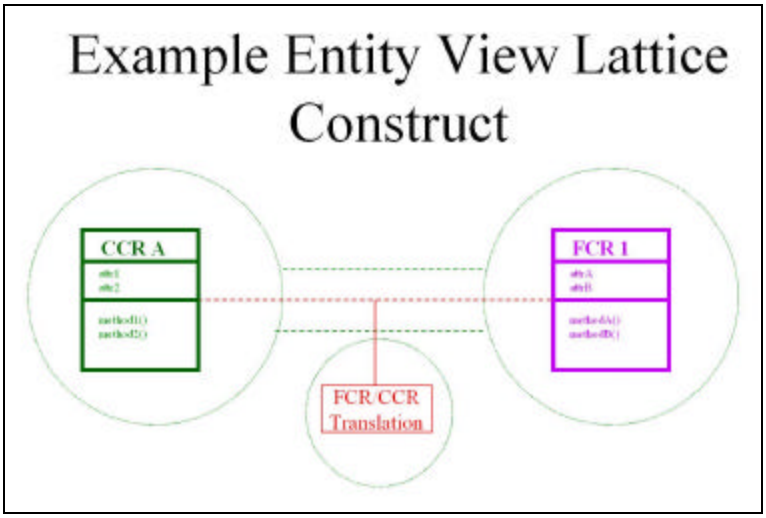

Figure IV-4. Example Entity View Lattice Construct.

The second lattice construct provides the FCR-to-FCR inheritance relationships. For Distributed OOMI we propose the use of UpCast and Down Cast Transforms to facilitate the generalization and specialization of FCRs. Figure IV-5 depicts the an FEV inheritance hierarchy relationship and again uses superimposed circles and dotted lines to suggest the appropriate generic containers and relationships.

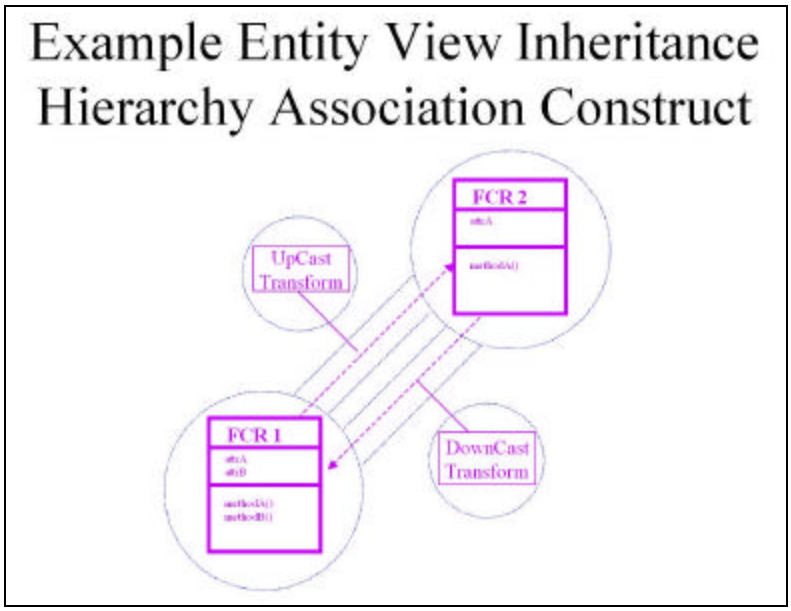

Figure IV-5. Example Entity View Inheritance Hierarchy Association Construct.

To generalize these two FIOM constructs for Distributed OOMI we simply remove the heavy-weight components while preserving the containers, relationships and behavior. Figure IV-6 depicts an abstraction of the OOMI FEV structure and Figure IV-7 the abstract lattice construct for inheritance hierarchy representation. Any FEV or inheritance relationship can be represented by the generic constructs by loading the appropriate heavy-weight components into the generic containers.

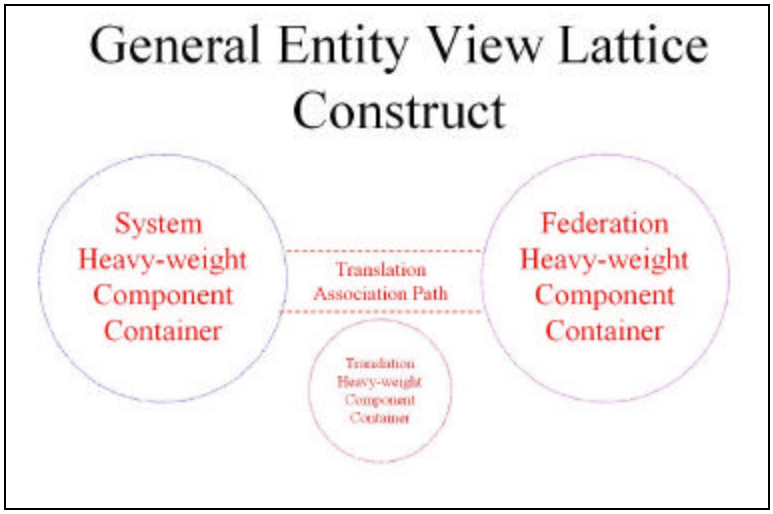

Figure IV-6. General Entity View FIOM Lattice Construct.

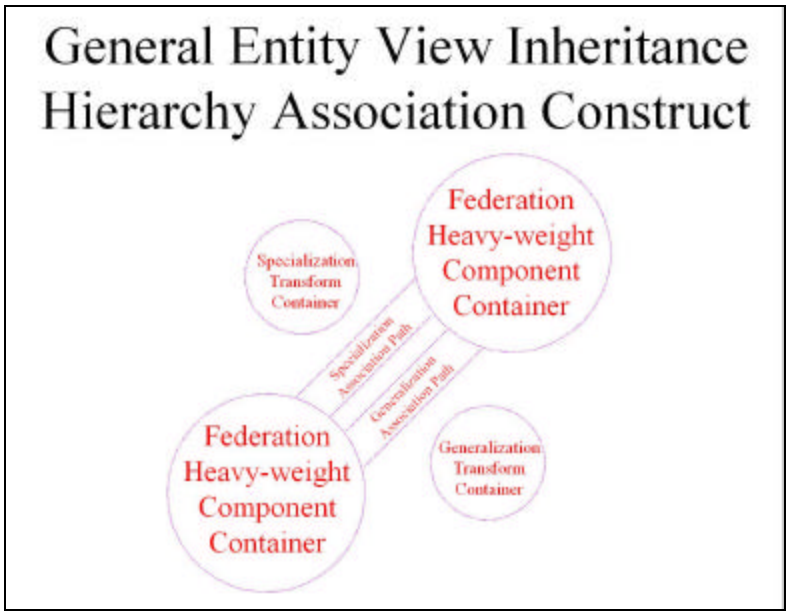

Figure IV-7. General Entity View Inheritance Hierarchy Association Construct.

By considering the general lattice constructs as object containers for the associated OOMI heavy-weight objects and associations, we are able to provide a model for the development of a data structure capable of generic lattice-wide behaviors, provided by a set of recurring *lattice types*. Combinations of these lattice constructs create the Local storage structure, notionally presented in Figure IV-8.

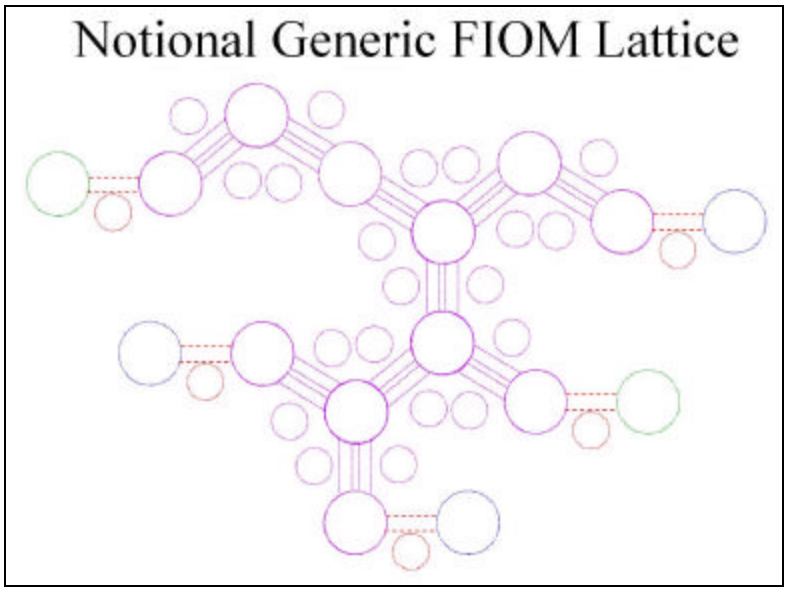

Figure IV-8. Notional Generic FIOM Lattice.

A Local storage container, such as the one shown in Figure IV-8 would be populated with FIOM heavy-weight components to produce a specific FIOM Lattice. The structure itself, however, is assembled using the generic lattice constructs, arranged according to the information stored in the FOIM light-weight components. In fact, the heavy-weight components need not be instantiated and positioned in the lattice structure until after the entire structure is assembled. Construction of such a lattice for an FIOM provides a scalable and abstract data structure into which functional capabilities, such as path discovery and path traversal can be incorporated.

3. Distributed OOMI Peer Networking Layer

The decision to include a Peer Networking layer in the architecture and to assume the availability of a suitable peer-to-peer implementation is based primarily on the findings of existing peer networking research efforts. For example, the SPAWAR Distributed Computing and Collaboration Framework (DCCF) research project focuses on efficient peer collaboration in a communication environment representative of actual naval information exchange systems, rather than the usual research assumption of ample link bandwidth [PD02]. Supplemental to DCCF are the peer networking advances from the civilian university research [BCM+02, BBK02], which, though not directly applicable to the DoD environment, demonstrate that the state-of-the-art in peer networking is advancing. When the decision is made to pursue distributed collaboration via peer networking, instead of pursuing the client/server based Web Services model, peer-to-peer implementations will be rapidly achieved from the existing base of peer networking research. Regardless, we recommend implementing the collaboration layer of Distributed OOMI according to the peer-networking model.

Consider for completeness an implementation that approximates the Peer-Cloud behavior using a client/server model. Recall that the solution aims to reduce the complexity of underlying network infrastructure, an example of which is shown in Figure III-4. As shown in Figure IV-9, the client/server model abstracts the network complexity into a logical star configuration. To produce the desired logic bus behavior to support collaboration, the server must relay and broadcast for all participating client nodes. This presents a scalability limitation and introduces a single point for failure.

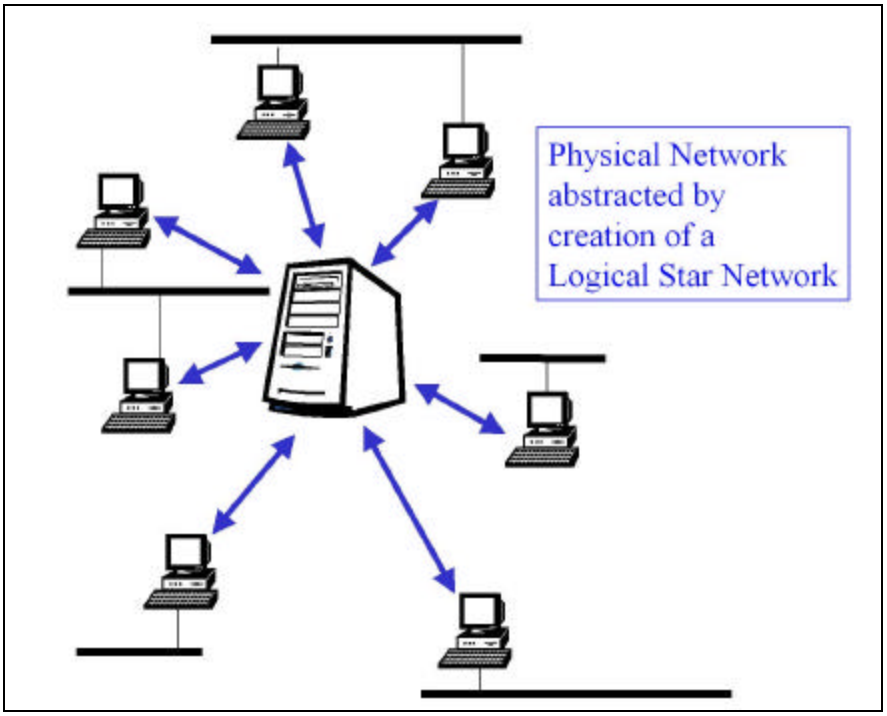

Figure IV-9. Example of a Logical Star Client/Server Model.

The client/server model is made scalable by introducing replicated servers to manage increasing loads. While potentially appropriate for the many-to-one relationship between Web clients and a Web provider, the client/server model is contrary to the many-to-many relationships necessary for distributed collaboration. The complexity of approximating logical bus behavior with a group of replicated servers unfortunately increases as a function of the level of replication. Additionally, the collaborative environment required for DoD Acquisition lacks a single, central organization or agency that can unilaterally provide the producer role for the client/server model.

Should it be necessary to use a client/server model, the best approach is to use the server to simplify organization of the peer network, without being directly involved in the peer-to-peer communications. An implementation such as this would use a server to act as the gateway for peers attempting to join the peer network. The server would organize the sparsely connected network proposed in Chapter III, as depicted in Figure IV-10. The red arrows in the figure represent communication with the server for the purpose of organizing the links of the sparse peer network. The blue arrows represent the peer

network links, via which peers collaborate. Recall from Chapter III, that the sparse network is further abstracted to create a logical bus.

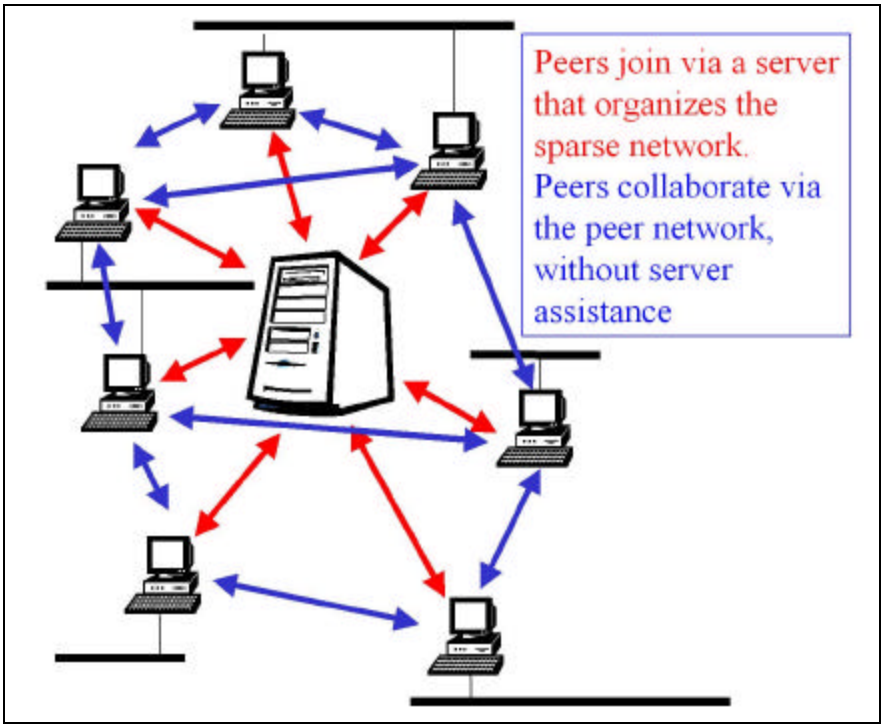

Figure IV-10. Example Peer Network Organized By Central Server.

Depending on the client/server model to form the peer network maintains the dependency on the server node, which represents a possible single point of failure. An improvement to the server-assisted organization of the sparsely connected peer network is possible by allowing any peer to perform the server's organization function, as depicted in Figure IV-11. Any peer can perform the organization task, and so a single peer is designated to act as the network organization server. Should this peer leave the peer network, another peer would assume the server responsibility.

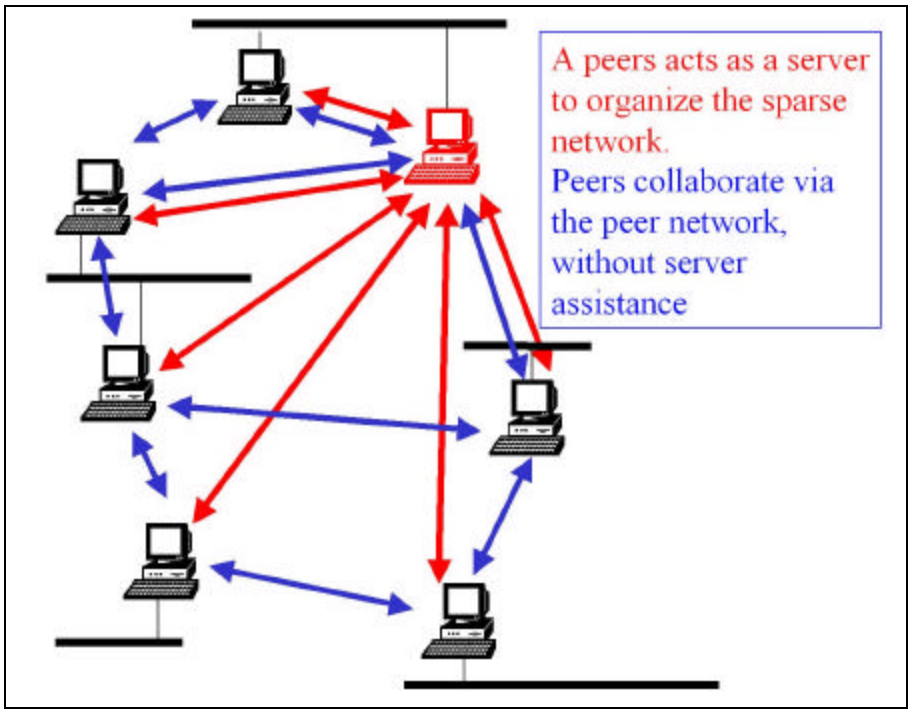

Figure IV-11. Sparse Peer Network Organized by Peer Server.

The University of Maryland research project [BBK02] is an example implementation that could easily be adapted to provide a server based peer network organization function. The project uses designated peer servers to establish a hierarchy for rapid multicast distribution of content [BBK02]. In their model, the server remains involved in the peer network communication, but the basic apparatus for designating a peer server, and the means for deciding the organization of a dynamic peer network are provided.

4. Connecting the Layers

Interaction between the layers is necessary, and as mentioned previously, should be designed to maintain modularity surrounding layer interfaces. The key interactions required by the architecture include: the addition of new FIOM components to Remote storage, the retrieval of components persistently held in Remote storage, the creation of the FIOM Lattice in Local storage, the re-direction of Storage layer requests to the Peer Cloud and the servicing of Peer-Cloud requests to the Storage layer.

The construction of an FIOM begins with the creation of the light-weight components that capture the logical containment relationships. The Federation Entity Manager module of the OOMI IDE facilitates this creation process as depicted in Figure IV-12.

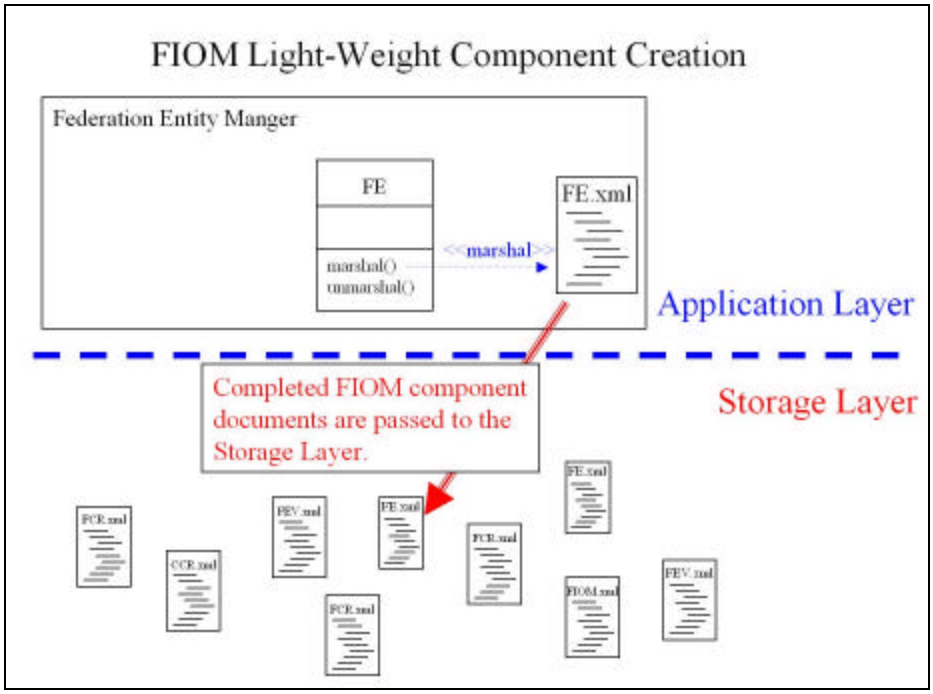

Figure IV-12. Example Of FIOM Light-Weight Component Creation.

A generic instance of the desired type of light-weight component is instantiated within the Federation Entity Manager. Based on user input and program logic, the generic class is populated with *component specific* information, such as a component name and references to other light-weight and heavy-weight components. After the user accepts the new light-weight component as complete, it is marshaled to XML. The resulting XML document is then passed to the Storage layer to be held persistently in Remote storage. Passing a component to Remote storage is of critical importance because of the *write once/read-only thereafter* policy of the Distributed OOMI Storage layer.

To use components held in Remote storage, the Federation Entity Manager requests the desired component by name from the Storage layer. The component is

passed to the Federation Entity Manager as an XML document. By instantiating a generic class instance for the light-weight component, the unmarshal method created by Castor is made available. The XML document is then unmarshaled into a class representing the specific light-weight component of interest. Figure IV-13 illustrates the process for an FIOM light-weight component. The name of the FIOM requested from the Storage layer is likely provided by the user. The sub components for the FIOM are held as references within the FIOM light-weight component.

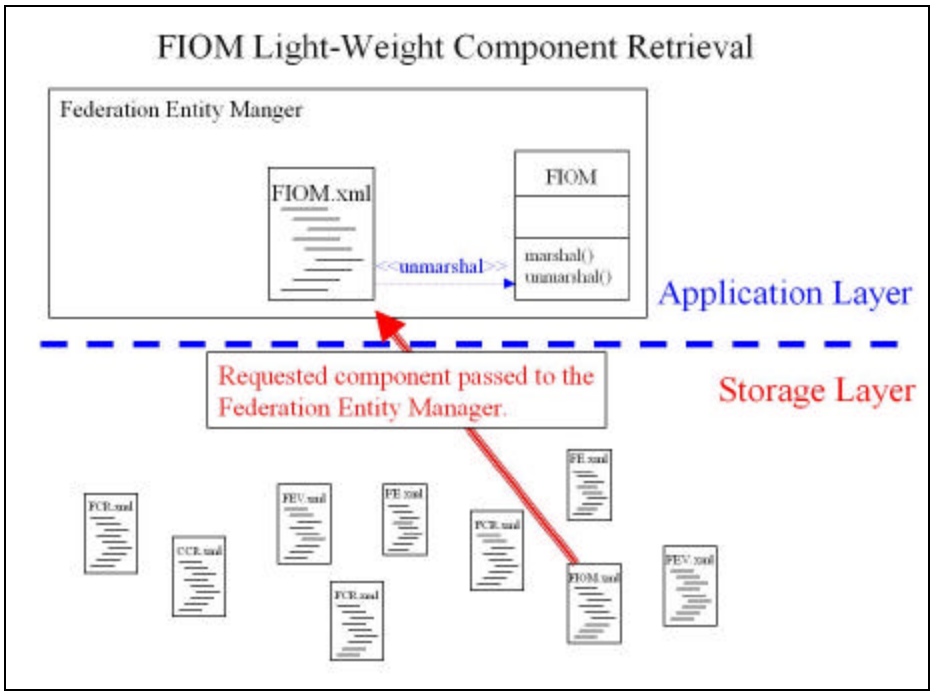

Figure IV-13. Example Of FIOM Light-Weight Component Retrieval For Use.

The FIOM will be constructed over time, however, and so we must provide for the inclusion of future sub-components within an FIOM light-weight component that is written only once. The future components need not be appended to the previously existing light-weight container component. Rather, a new component is constructed to contain the references to the new components. For example to add an FE to an FIOM, a new FIOM light-weight component is created that contains the reference to the new FE. The new FIOM component is simply given the same name as the previously created FIOM light-weight component. This does not cause a collision at the Storage layer, because Remote storage decouples the component name from the storage file name.

Many light-weight components referencing the same FIOM construct can be placed in remote storage by simply altering the file names used to persistently write the XML files to physical storage. The addition of a date-time stamp to the name of the component is one example of a simple method of obtaining a unique file name for Remote storage components that reference the same FIOM light-weight component. During the retrieval process the Federation Entity Manager uses the light-weight components it re-instantiates as Java objects to populate the data structures of the other IDE modules, as well as to build the Local storage lattice structure. By simply iterating over the documents provided by Remote storage until all the documents have been processed, we provide a scalable and extensible way to allow an FIOM to expand over time.

Requests to the Storage layer that cannot be satisfied by Remote or Local storage are re-directed to the Peer Networking layer, and therefore to the Peer Cloud. In non-real-time, the Peer Cloud directs the request to all participating peers and returns any valid replies to the Storage layer. New components can be *pushed* to the Peer Cloud by simply passing the component from the Storage layer to the Peer Networking layer. The complete scheme for peers to decide to either *cache*, *store* or *discard* components received via the Peer Cloud is not entirely known, however these three actions define the scope of peer response. Caching by the Peer Networking layer indicates a temporary holding of a component, perhaps at the Peer Networking layer level, and could be used to facilitate the desired multicast and relay behavior for Peer Cloud requests and replies. Storing indicates passing the component in question to the Storage layer of the host in question. This action causes the component to be placed in Remote storage, in effect replicating the component throughout the Peer Cloud. Discarding of a component means no action concerning the component is taken by the host. The store and discard actions can be used together to cause a component pushed to the peer network to be stored by the first *n* peers that encounter the component, and discarded thereafter. For example, new components could be quickly replicated in the Peer Cloud by pushing them to the first three (n=3) peers in the network for storage. Additionally, any peer that specifically requests a component from the Peer Cloud will store any and all replies, providing further replication of the FIOM components. Initially, the entire FIOM may be available via the

Storage layer at every participating peer. Over time, though, as the number of peers increase and the FIOM increases in size, the structure will grow at the edges, causing the complete set of components required to construct the entire FIOM to be distributed between the participants according to the specific section of the Federation with which a participant's component system relates. Pushing the replication of components several peers deep into the Peer Cloud improves the failure handling and error recovery by providing overlap in FIOM component Remote storage among the peers. Pushing heavy-weight components to the Peer Cloud is even more important than pushing light-weight components, because the architecture causes heavy-weight components to be explicitly requested from the Peer Cloud less frequently than light-weight objects, thereby reducing the amount of replication of heavy-weight components, as originally intended.

D. IMPLEMENTATION SUMMARY

The hurdles to realizing an implementation of Distributed OOMI are still considerable, but not insurmountable. The suggestions of this implementation chapter are practically a list of future research items, particularly the FIOM Lattice construction. In summary, the overarching lesson from implementing the prototype OOMI IDE and portions of a Distributed OOMI IDE is that layering and modularity are valuable, but that centralized data storage is not conducive to good layering and effective modularity. Each portion of an implementation has specific data requirements and should, in true object-oriented fashion, handle those requirements internally, and specifically to the delegated task. Monolithic data storage should be avoided in distributed applications, and the fear of replicating specific data in more than one storage location must be alleviated to meet the requirements of a distributed environment and to leverage the advantage of a distributed system. Physical storage is inexpensive and as long as data structures adhere to the combined mantra of 'self-similarity with load distribution according to a Power Law,' the storage solution will scale, and the inevitable replication of data in many locations will slow as the system becomes very very large, and eventually the storage load will be spread evenly across the entire system.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS

This Chapter summarizes the benefit of this research effort by describing a vision of the future made possible by the OOMI. We revisit the reasoning for pursuing the Distributed OOMI and argue that the results presented in this Thesis support the DoD environment. Open issues for future research are enumerated and described briefly as well.

A. OOMI IN THE RUNTIME ENVIRONMENT

The concepts from this research are primarily intended for the pre-runtime FIOM construction activity, although the concepts that potentially apply equally to the runtime environment are: peer networking, self-similar storage constructs and FIOM Lattice. The Peer Cloud generated by creating a logical bus abstraction on top of a peer-to-peer networking capability provides the ability to define contexts for network communications, and then group communications by context. The self-similar storage constructs are the key to a collaborative sharing environment that scales well. Finally the FIOM Lattice structure provides the means to support the use of runtime OOMI Translators in resolving differences in representation and view.

The ability to implicitly use context to decipher communication is paramount to human visual and auditory capabilities, but has never been fully developed for data communications. The de facto Internet standard TCP/IP suite contributes considerably to this phenomenon because of the tight coupling between TCP and IP. By defining a data communication *session* to be applicable to the discrete transfer of data from one point to another via a single connection protocol, the ability to represent contextual communication is severely limited under TCP/IP. Peer overlay networks provide a layer between TCP/IP and the application, in which context can be established, context can persist and access to numerous communication protocols using numerous connections is possible. Currently the user provides these sorts of desirable capabilities. Tactical military communications require the same notion of context, and as with the current Internet, context is provided by users. In years past context in military communications

could only be provided based on human input, because the entire network of communications circuits required heavy human intervention to configure and control. The future communication environment will resemble the packet switching capabilities of the Internet and the circuit switching capabilities of the telephone networks. Support for the creation of peer-to-peer network overlays for the military tactical networks is one way to provide context to the communications and promote the concepts of Network Centric Warfare.

The use of self-similar data structures to provide scalable storage solutions for complex data modeling has a direct parallel application to military systems. The common design view of an information system under development considers the data to be the most constant and consistent portion of the system design; the interfaces are considered prone to incremental change over time, and the user requirements for the system are thought to be in constant flux and thus never achievable. By reversing this notion, the operational view of a system is represented, in which the operational requirements for the system are fixed, the interface can and will change over time, and the data is in constant flux. Self-similar data structures scale to allow a system to respond to changing data requirements and in order to meet the fixed operational system requirements. The reason the operational view of a system is more correct than the common design view is that any automation system is dealing with data in order to ultimately model some real-world entity. In modeling a real-world entity, abstractions must be used, causing portions of the real-world entity to be approximated or completely ignored by the system designers. The real-world entities that are of interest, and hence modeled in data by automated systems, are dynamic. The data abstractions used to represent a dynamic entity must therefore be supplemented periodically with new data to accurately develop a representation of the current state of the entity. When the system is designed based on the notion that the data is fixed, static data structures are developed that represent in abstraction dynamic entities. New data supplied to a system with static data structures is not supplemental, but rather a refresh, generating a sequence of snapshots of the real-world entity, rather than a data model that converges toward reality.

Self-similar data structures scale to allow dynamic representation of dynamic real-world entities.

The FIOM Lattice scales the FIOM to fit the runtime environment in which the OOMI Translator is deployed. The FIOM Lattice facilitates the customized assembly of sub-sections of an FIOM, sized to meet the requirements of a particular interoperability runtime context. The complete FIOM is essentially viewed at any number of levels of abstraction by constructing an FIOM Lattice containing only the constructs and lattice paths that relate a specific sub set of FIOM registered component systems. The FIOM Lattice is an extensible and scalable construct, so that the level of abstraction represented by a lattice is dynamic as a function of the needs of the OOMI Translator environment.

B. INTEROPERABILITY BY DESIGN

Interoperability between systems is not, as commonly suspected, concerned with aggregating existing system capabilities to provide better access to more, static data, thereby providing a means with which to meet changing user requirements. Interoperability is information exchange and joint task execution in order to improve the ability to supplement dynamic models of real-world entities by aggregation of available data from any number of sources, each of which operates under fixed requirements. There exists a moving target in interoperability; however, it is not the user requirements for the system. The moving target for interoperability is the data that represents the most current and complete view of the real-world entities and this data is not stored monolithically to be accessed; it must be constantly collected by observation of real-world entities. The system interfaces are the middle ground and the best position from which to approach the solution. The solution cannot be in terms of system requirements and top-down standards, though. It must deal instead with resolving the differences in modeling that prohibit information exchange and joint task execution.

1. The Case for 'Legacy' Systems

A great strength of the United States Government is the inherent ability to distribute effort at every level and yet pursue, for the most part, common goals. The DoD acquisition structure is distributed in order to produce stability and locally optimal solutions for locally specific problems. A common misconception is that there is a path

to interoperability based on a flat acquisition structure, in which coordinated development can eliminate the existence of stove-piped systems by enforcing unilateral standards and unilateral system requirements. The fallacy of this viewpoint is the assumption that system requirements are the cause of the interoperability problem and that using standards will save money. Modeling and abstractions that lead to static data structures are the real reason for the inherent lack of forward-compatibility that earns systems the title 'legacy.' The design choices that cause system data modeling to be the problem are made in order to locally optimize the system with respect to cost, performance and schedule, while meeting system design requirements. Unilaterally mandating or standardizing system data structures, system design requirements, or even system interfaces, forces design decisions that are non-locally optimal with respect to cost, performance and or schedule.

2. Bottom-Up Standard

The OOMI develops a standard intermediate representation for interoperability, but does not interfere with the local optimization and distribution strengths of the DoD acquisition structure. An FIOM effectively collects the results of the locally optimal design decisions for groups of 'legacy' systems and resolves the modeling differences, thereby defining the standard for system interoperability one system at a time, from the bottom up. No system is restricted with respect to optimizations during the design and development, and yet no system is excluded from the potential benefits of information exchange and joint task execution.

Generating a standard from the bottom up is difficult, but not impossible. The first step is to create a collaborative environment that spans the distributed base of DoD's hierarchical acquisition structure, and allows locally optimal participation via an asynchronous FIOM construction effort. The Internet provides the ubiquitous connectivity that allows for the creation of a horizontal collaboration plane that virtually unites all the existing knowledge concerning the component systems to be registered in the FIOM. Peer networking via the Internet adds context to the collaborative environment, and the self-similar data structures introduced in Chapter III allow the FIOM to persist as well as be extensible. Figure V-1 depicts the logical concept of the

collaborative environment by including the Program Offices and Contractors within the shaded box. The blue ellipse depicts the FIOM, encompassing the target systems and resolving their differences in representation and view, producing a bottom-up standard for information exchange and joint task execution.

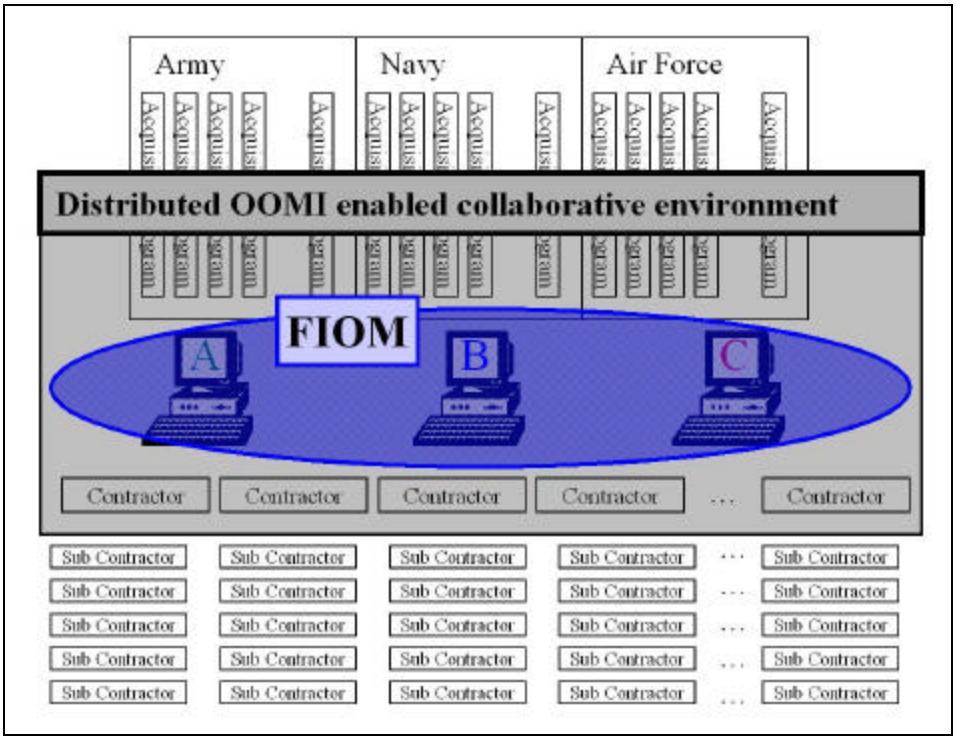

Figure V-1. FIOM Construction Enabled By Collaborative Environment.

C. OPEN ISSUES FOR THE FUTURE

1. If the Solution Fits, Wear It

The converse of this section's heading is something to the effect of; 'if a solution is not worn, it does not fit.' Of the existing technologies and frameworks available that are held up as solutions to interoperability concerns, none are presently used in the resolution of interoperability issues between DoD legacy systems. The reason is not that the proposed solutions are simply unfashionable or too expensive, but that the promise of COM, DCOM, COM+, CORBA, HLA, SOAP, Web Services, etc…(insert your favorite cutting edge buzz-word or technology), remains unrealized because of the respective inherent inadequacies of the approach. Of primary concern is that lack of attention afforded the development environment in which the interoperability solution must be

applied to achieve the desired result. The overwhelming tendency is to develop a solution that applies to a homogeneous and non-distributed environment and then attempt to extend the solution to fit the real world distributed requirements. A distributed environment requires a distributed solution and it appears that the nature of the environment in which DoD interoperability can be solved is incompatible with the environments in which the majority of proposed solutions are designed to function. This realization leads to our short list of suggestions for future research and open issues for Distributed OOMI, which are: (1) research into the existence of additional distributed systems paradigms, (2) research into the concept of using a lattice to represent the OOMI FEV inheritance hierarchy, and (3) the development and implementation of a peer networking solution for Distributed OOMI.

2. Distributed OOMI as a Collaborative Solution

Our first recommendation for future research stems from the concern that interoperability is a distributed computing problem and thus would benefit from an effort to discover additional distributed systems paradigms and the characteristics that describe the boundaries of these paradigms. We set out in this research effort to describe an architecture to guide a design and development effort toward the full implementation of the OOMI. The result is a blue print for a distributed system, which, according to the classification paradigms introduced in Chapter II, is unlike other distributed systems research efforts. The hypothesis for future research then is: Does the architecture presented for the Distributed OOMI succeed in describing a collaborative solution to the problem of sharing among distributed nodes, and does, therefore, a distributed systems paradigm exist that captures the properties of collaborative environments in general and according to which similar collaborative environment construction efforts may be compared? We feel that the answer to both questions is yes, and that the paradigm model may be as depicted in Figure V-2.

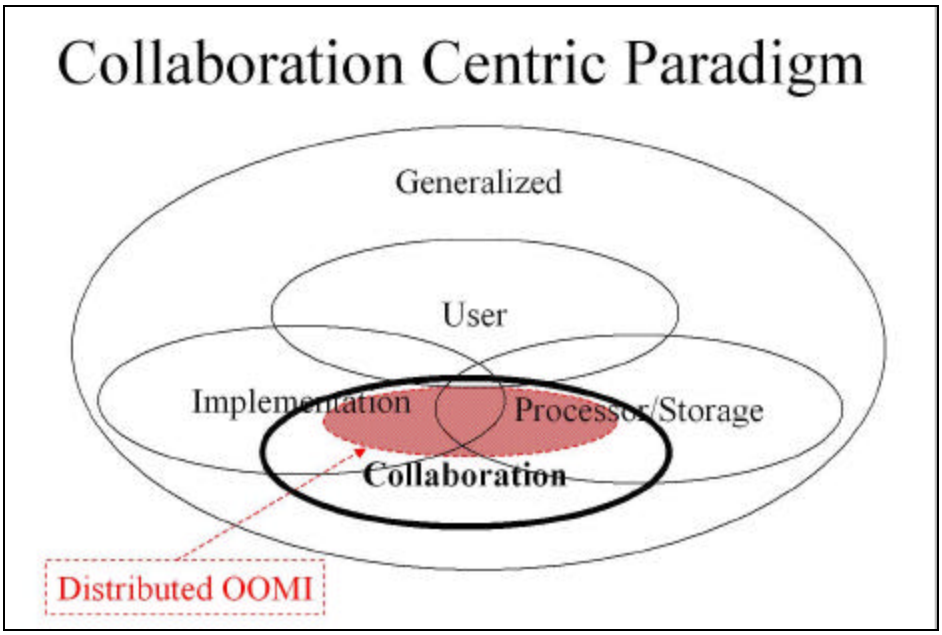

Figure V-2. Distributed OOMI And The Collaboration Centric Paradigm.

3. Use of Lattice Structure for Representing Inheritance Relationships Among FIOM Components

The idea for the FIOM Lattice grew out of frustration with trying to implement the FEV inheritance hierarchy simply without compounding the difficulty with which the FIOM could be accessed and used in support of the OOMI IDE. The primary issues of the lattice concept for FIOM FEV inheritance hierarchy are: (1) deciding on the correctness and completeness of the use of UpCast and DownCast transforms for generalization and specialization of FCR Schema classes, (2) the definition and design of the general lattice constructs that facilitate the scalability of FIOM Lattice structure, and (3) the consideration of the container for an FIOM Lattice, including the functional capabilities of the container to construct an empty lattice, populate a lattice, search a lattice, traverse paths in a lattice, extract information from a lattice and facilitate OOMI Translator operation.

4. Implementing Peer Networking in Distributed OOMI

The peer networking component of the Distributed OOMI architecture introduces several open issues, which are; (1) investigating the completeness and correctness of using a peer network to create a logical bus network, (2) investigating the use of a logical

bus network to enable context based communication for collaboration, and (3) the development of the mechanism for creation of a dynamic and sparsely connected peer network on top of the existing TCP/IP based Internet.

A complete implementation of an OOMI IDE remains an open issue. With the introduction of the Distributed OOMI architecture, the design and development of an OOMI IDE are made simpler in some respects and more complex in others. The IDE remains valuable for proof of OOMI concepts and is therefore a worthwhile effort.

D. SUMMARY

There remains much to do in both the world of interoperability and within the scope of the OOMI. We've suggested many additions and improvements to the OOMI IDE implementation effort. We've introduced and discussed the problem of interoperability in the DoD context. We've shown the impact of the distributed DoD acquisition environment and developed a solution that addresses the unique requirements of this environment and can summarize it all with a simple graphic depicting the road ahead. Figure V-3 depicts the realization that interoperability can only be solved if a collaborative environment conducive to producing the solution is developed first.

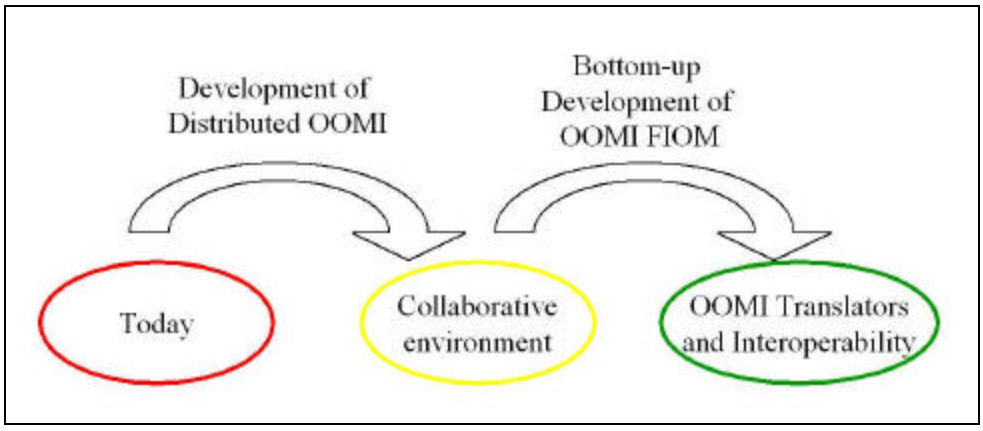

Figure V-3. The Path To Interoperability.

DoD's present distributed structure and substantial base of existing non-interoperable systems is represented by the 'today' elipse in Figure V-3. Interoperability for DoD's multitude of systems is shown to be at least two steps away and unfortunately, the phrase 'we can't get there from here' accurately describes the current state of affairs.

We can describe what interoperability means and how to provide the capability to exchange information and jointly execute tasks by defining a bottom-up standard for interoperability according to Young's OOMI. This is the finaly step to enabling interoperability. The intractable portion of using the OOMI is the distributed environment from which the required FIOM will be generated. We can develop a distributed system, however, that produces a horizantal collaboration capability on top of the current DoD structure and that facilitates the distributed collaboration in the construction on an FIOM.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[AAC+01] Ahmed, K., Ancha, S., Cioroianu, A., et al. *Proffesional Java XML*, Wrox Press, Birmingham, UK, 2001.

[Alm98] Almeida, J., *Software Architecture for Distributed Real-Time Embedded Systems*, Master's thesis, Naval Postgraduate School, Monterey, California, September 1998.

[BBK02] Banerjee, S., Bhattacharjee, B., and Kommareddy, C. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM 2002 Conference*. ACM, (New York, New York, August 2002), 205-217.

[BCM+02] Byers, J., Considine J., Mitzenmacher, M., et. al. Informed Content Delivery Across Adaptive Overlay Networks. In *Proceedings of ACM SIGCOMM 2002 Conference*. ACM, (New York, New York, August 2002), 47-60.

[Bla01] Blandin, A. "Castor XML Source Code Generator, User Document.", [http://www.castor.org/SourceGeneratorUser.pdf], July 2001, accessed January 2003.

[BM01] Biron, P., Malhotra, A. (editors), "XML Schema Part 2: Datatypes", [http://www.w3.org/TR/xmlschema-2/], May 2001, accessed August 2002.

[CDK01] Coulouris, G., Dollimore J., Kindberg T. *Distributed Systems Concepts and Design*. 3rd ed. Harlow, England: Addison-Wesley, 2001.

[CKB+00] Carcalho, D., Kon, F., Ballesteros, F., Roman, M., Campbell, R. and Mickunas, D. Management of Environments in 2K. In *Proc. Seventh Int. Conf. Parallel and Distributed Systems*, IEEE (Iwate, Japan, July 2000), 479-485.

[Chr01] Christie, B., *Integrated Development Environment(IDE) for the construction of a Federation Interoperability Object Model(FIOM).* Master's Thesis, Naval Postgraduate School, Monterey, California, September 2001.

[FF02] Faloutsos, M., and Faloutsos, C. "Data Mining the Internet, What We Know, What We Don't Know and How We Can Learn More." Tutorial at ACM SIGCOMM Conference, Pittsburgh, Pennsylvania, 20 August, 2002.

[FK98] Foster, I., Kesselman, C. The Globus Project: A status report. In *Proc. Seventh Heterogeneous Computing Workshop*, IEEE (Orlando, Florida, Mar. 1998), 4-18.

[Gra98] Grand, Mark., *Patterns in Java,* Volume 1, Wiley Computer Publishing, New York, New York, 1998.

[Gra98] Grand, Mark., *Patterns in Java,* Volume 2, Wiley Computer Publishing, New York, New York, 1998.

[GW97] Grimshaw, A.S., Wulf, W.A. The Legion vision of a worldwide virtual computer. *Comm. ACM* 40,1 (January 1997), 39-45.

[HCD+01] Hunter, D., Cagle, K., Dix, C., Kovack, R., Pinnock, J., Rafter, J., *Beginning XML*, 2ed, Wrox Press, Birmingham, UK, 2001.

[HH02] Le Hors, A., Le Hégaret, P. (editors), "Document Object Model (DOM) Level 3 Core Specification", [http://www.w3c.org/TR /2002/WD-DOM-Level-3-Core-20020409/], 4 April 2002.

[HM00] Hunter, J., McLaughlin, B., "JDOM Frequently Asked Questions", [http://www.jdom.org/], April 2000, accessed August 2002.

[KCM+00] Kon, F., Campbell, R.H., Mickunas, M.D., Nahrstedt, K. and Ballesteros, F.J. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the Ninth Internat. Symposium on High-Performance Distributed Computing*, IEEE (Pittsburgh, Pennsylvania, Aug 2000), 201-208.

[Kre99] Kreeger, G., *Requirements Analysis and Design of a Distributed Architecture for the Computer Aided Prototyping System (CAPS)*, Master's thesis, Naval Postgraduate School, Monterey, California, September 1999.

[KR01] Kurose, J., Ross, K., *Computer Networking a Top-Down Approach Featuring the Internet*, Addison Wesley, Boston, Massachuesettes, 2001.

[Lan99] Laundauer, T., *The Trouble with Computers*, MIT Press, Cambridge, Massachuesetts, 1999.

[Lee02] Lee, S., *Class Translator for the Federation Interoperability Object Model (FIOM*), Master's thesis, Naval Postgraduate School, Monterey, California, March 2002.

[LW94] Liskov, B., Wing, J., "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems,* Vol. 16, No. 6, November 1994, pp. 1811-1841.

[ML02] Michael, J., and Lawler, G., "Classification Paradigms for Comparing Distributed Systems," paper submitted for publication in *Communications of the ACM*, Naval Postgraduate School, Monterey, California, October 2002.

[PD02] Putnam, C.D., Duffy, L., "Distributed Computing and Collaboration Framework (DCCF)", Space and Naval Warfare Systems Center, San Diego, California, October 2002.

[Pre01] Pressman, R., *Software Engineering A Practitioner's Approach*, 5th ed., McGraw Hill, Boston Massachuesettes, 2001.

[Pug01] Pugh, R.G., *Methods For Determining Object Correspondence During System Integration*, Master's Thesis, Naval Postgraduate School, Monterey, California, 2001.

[She02] Shedd, S.F., *Semantic and Syntactic Object Correlation in the Object-Oriented Method for Interoperability*, Master's Thesis, Naval Postgraduate School, Monterey, California, 2002.

[SM99] Skousen, A. and Miller, D. Using a single address space operating system for distributed computing and high performance. In *Proc. Eighteenth Annual Int. Performance, Computing, and Communications Conf.*, IEEE (Scottsdale, Arizona, February 1999),8-14.

[SM00] Skousen, A. and Miller, D. The Sombrero single address space operating system prototype: A test bed for evaluating distributed persistent system concepts and implementation. In *Proc. Internat. Conference on Parallel and Distributed Processing Techniques and Applications*, CSREA Pres, (Las Vegas, Nevada, June 2000), 557-563.

[Sta00] Stallings, W., *Data and Computer Communications*, 6th ed., Prentice Hall, Upper Saddle River, New Jersey, 2000.

[Sun01] Sun Microsystems, Inc. (publisher), "The Java Tutorial", [http://java.sun.com/docs/books/tutorial/] update September 2001, accessed October 2001.

[TBM+01] Thompson, H., Beach, D., Maloney, M., Mendelsohn, N. (editors), "XML Schema Part 1: Structures", [http://www.w3.org/TR/xmlschema-1/], May 2001, accessed August 2002.

[Vau02] Vaughan-Nichols, Steven J. Developing the Distributed Computing OS. In *Computer*, IEEE (Los Alamitos, California, September 2002), 19-21.

[WAL98] Wu, D., Agrawal, D., and El Abbadi, A., StratOSphere: Mobile processing of distributed objects in Java. In *Proc. Fourth Annual ACM/IEEE Int.. Conf. Mobile Computing and Networking,* ACM (Dallas, Texas, October 1998), 121-132.

[WO00] Wing, J.M., "Respectful Type Converters", *IEEE Transactions on Software Engineering*, Volume 26, Number 7, July 2000, pp. 579-593.

[You02] Young, P.E., *Heterogeneous Software System Interoperability Through Computer-Aided Resolution of Modeling Differences*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, 2002.

APPENDIX A

A. LIGHT-WEIGHT FIOM COMPONENTS:

We include the following XML Schema as examples of the intended scope and purpose of the light-weight component data structures introduced in Section III.D.4. Perhaps contrary to current convention with XML Schema, we separate the individual light-weight components into separate XML Schema definitions, rather than combining the definition of light-weight component element tag sets in a single XML Schema document. The separate XML Schema definitions decouple the light-weight components and simplify the data binding process that is used to generate specific Java classes for each type of light-weight component.

1. XML Schema For An FIOM Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
       <xs:element name="fiom">
              <xs:annotation>
                     <xs:documentation>
                            Federation Interoperability Object Model
                     </xs:documentation>
              </xs:annotation>
              <xs:complexType>
                     <xs:sequence minOccurs="0" maxOccurs="unbounded">
                            <xs:element name="federationEntityLink">
                                   <xs:complexType>
                                          <xs:attribute name="name" type="xs:string"
                                             use="required"/>
                                   </xs:complexType>
                            </xs:element>
                     </xs:sequence>
                     <xs:attribute name="componentOf" type="xs:string" use="required"/>
                     <xs:attribute name="name" type="xs:string" use="required"/>
                     <xs:attribute name="authority" type="xs:string" use="required"/>
              </xs:complexType>
       </xs:element>
</xs:schema>
```

2. XML Schema For An FE Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
       <xs:element name="fe">
              <xs:annotation>
                     <xs:documentation>
                            Federation Entitiy
                     </xs:documentation>
              </xs:annotation>
              <xs:complexType>
                     <xs:sequence minOccurs="0" maxOccurs="unbounded">
                            <xs:element name="entityViewLink">
                                   <xs:complexType>
```

```
                                        <xs:attribute name="name" type="xs:string"
                                           use="required"/>
                                    </xs:complexType>
                                </xs:element>
                            </xs:sequence>
                            <xs:attribute name="componentOf" type="xs:string" use="required"/>
                            <xs:attribute name="name" type="xs:string" use="required"/>
                            <xs:attribute name="authority" type="xs:string" use="required"/>
                    </xs:complexType>
            </xs:element>
</xs:schema>
```

3. XML Schema For An FEV Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
        <xs:element name="fev">
                <xs:annotation>
                        <xs:documentation>
                                Federation Entity View
                        </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                        <xs:sequence minOccurs="1">
                                <xs:element name="fcrLink">
                                        <xs:complexType>
                                                <xs:attribute name="name" type="xs:string"
                                                   use="required"/>
                                        </xs:complexType>
                                </xs:element>
                                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                        <xs:sequence>
                                                <xs:element name="ccrLink">
                                                        <xs:complexType>
                                                                <xs:attribute name="name"
                                                                   type="xs:string"
                                                                   use="required"/>
                                                        </xs:complexType>
                                                </xs:element>
                                                <xs:element name="translationLink">
                                                        <xs:complexType>
                                                                <xs:attribute name="name"
                                                                   type="xs:string"
                                                                   use="required"/>
                                                        </xs:complexType>
                                                </xs:element>
                                        </xs:sequence>
                                </xs:sequence>
                                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                        <xs:element name="upCastLink">
                                                <xs:complexType>
                                                        <xs:attribute name="name"
                                                           type="xs:string" use="required"/>
                                                </xs:complexType>
                                        </xs:element>
                                </xs:sequence>
                                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                        <xs:element name="downCastLink">
                                                <xs:complexType>
                                                        <xs:attribute name="name"
                                                           type="xs:string" use="required"/>
                                                </xs:complexType>
                                        </xs:element>
                                </xs:sequence>
                            </xs:sequence>
                            <xs:attribute name="componentOf" type="xs:string" use="required"/>
                            <xs:attribute name="name" type="xs:string" use="required"/>
                            <xs:attribute name="authority" type="xs:string" use="required"/>
```

```
            </xs:complexType>
        </xs:element>
</xs:schema>
```

4. XML Schema For An FCR Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
        <xs:element name="fcr">
            <xs:annotation>
                <xs:documentation>
                    Federation Class Representation
                </xs:documentation>
            </xs:annotation>
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="fcrSchema">
                        <xs:complexType>
                            <xs:attribute name="name" type="xs:string"
                               use="required"/>
                            <xs:attribute name="package"
                               type="xs:string" use="required"/>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="fcrSyntax" minOccurs="0">
                        <xs:complexType>
                            <xs:attribute name="name" type="xs:string"
                               use="required"/>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="fcrSemantics" minOccurs="0">
                        <xs:complexType>
                            <xs:attribute name="name" type="xs:string"
                               use="required"/>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="componentOf" type="xs:string" use="required"/>
                <xs:attribute name="name" type="xs:string" use="required"/>
                <xs:attribute name="authority" type="xs:string" use="required"/>
            </xs:complexType>
        </xs:element>
</xs:schema>
```

5. XML Schema For A CCR Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
        <xs:element name="ccr">
            <xs:annotation>
                <xs:documentation>
                    Component Class Representation
                </xs:documentation>
            </xs:annotation>
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="ccrSchema">
                        <xs:complexType>
                            <xs:attribute name="name" type="xs:string"
                               use="required"/>
                            <xs:attribute name="package"
                               type="xs:string" use="required"/>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="ccrSyntax" minOccurs="0">
                        <xs:complexType>
                            <xs:attribute name="name" type="xs:string"
```

```
                                        use="required"/>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="ccrSemantics" minOccurs="0">
                                <xs:complexType>
                                        <xs:attribute name="name" type="xs:string"
                                           use="required"/>
                                </xs:complexType>
                        </xs:element>
                </xs:sequence>
                <xs:attribute name="componentOf" type="xs:string" use="required"/>
                <xs:attribute name="name" type="xs:string" use="required"/>
                <xs:attribute name="authority" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

6. XML Schema For An FCR-To-CCR Translation Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
    <xs:element name="translation">
        <xs:annotation>
            <xs:documentation>
                Translation between CCR and FCR
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence minOccurs="0">
                <xs:element name="class">
                    <xs:complexType>
                        <xs:attribute name="name" type="xs:string"
                           use="required"/>
                        <xs:attribute name="package"
                           type="xs:string" use="optional"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="uml">
                    <xs:complexType>
                        <xs:attribute name="name" type="xs:string"
                           use="optional"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute name="componentOf" type="xs:string" use="required"/>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="authority" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

7. XML Schema For An UpCast Transform Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
    <xs:element name="upCast">
        <xs:annotation>
            <xs:documentation>
                Translation between FCR and FCR
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence minOccurs="0">
                <xs:element name="class">
                    <xs:complexType>
                        <xs:attribute name="name" type="xs:string"
                           use="required"/>
```

```
                                        <xs:attribute name="package"
                                           type="xs:string" use="optional"/>
                                    </xs:complexType>
                                </xs:element>
                                <xs:element name="uml">
                                    <xs:complexType>
                                        <xs:attribute name="name" type="xs:string"
                                           use="optional"/>
                                    </xs:complexType>
                                </xs:element>
                            </xs:sequence>
                            <xs:attribute name="componentOf" type="xs:string" use="required"/>
                            <xs:attribute name="name" type="xs:string" use="required"/>
                            <xs:attribute name="authority" type="xs:string" use="required"/>
                        </xs:complexType>
                </xs:element>
        </xs:schema>
```

8. XML Schema For A DownCast Transform Component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by George Lawler (NPS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
        <xs:element name="downCast">
                <xs:annotation>
                        <xs:documentation>
                                Translation between FCR and FCR
                        </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                        <xs:sequence minOccurs="0">
                                <xs:element name="class">
                                        <xs:complexType>
                                                <xs:attribute name="name" type="xs:string"
                                                    use="required"/>
                                                <xs:attribute name="package"
                                                    type="xs:string" use="optional"/>
                                        </xs:complexType>
                                </xs:element>
                                <xs:element name="uml">
                                        <xs:complexType>
                                                <xs:attribute name="name" type="xs:string"
                                                   use="optional"/>
                                        </xs:complexType>
                                </xs:element>
                        </xs:sequence>
                        <xs:attribute name="componentOf" type="xs:string" use="required"/>
                        <xs:attribute name="name" type="xs:string" use="required"/>
                        <xs:attribute name="authority" type="xs:string" use="required"/>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Valdis Berzins
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

4. CAPT Paul Young USN
   Department of Computer Science
   United States Naval Academy
   Annapolis, Maryland

5. Professor Bret Michael
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

6. Mr. Michael Seebold
   Vice President of Corporate Operations
   Saffron Technology
   Morrisville, North Carolina

7. LT George Lawler USN
   SUPSHIP Portsmouth
   Norfolk, Virginia